out-of-core extension of the MUMPS solver

Abdou Guermouche, Labri Bordeaux

MUMPS Users Group Meeting, April 2010
Solving sparse linear systems

\[ Ax = b \]
⇒ Direct methods: \( A = LU \)

Typical matrix: BRGM matrix

- 3.7 \( \times 10^6 \) variables
- 156 \( \times 10^6 \) non zeros in \( A \)
- 4.5 \( \times 10^9 \) non zeros in \( LU \)
- 26.5 \( \times 10^{12} \) flops
Solving sparse linear systems

\[ Ax = b \]

⇒ Direct methods: \( A = LU \)

Typical matrix: BRGM matrix

- \( 3.7 \times 10^6 \) variables
- \( 156 \times 10^6 \) non zeros in \( A \)
- \( 4.5 \times 10^9 \) non zeros in \( LU \)
- \( 26.5 \times 10^{12} \) flops
Context

Physical constraint
- Core memory
- Memory required
- Memory crash

Software challenge
- Implementation of an out-of-core execution scheme within MUMPS
Out-of-core

Core memory  Disks

Memory required

Use of disks

Software challenge

- Implementation of an out-of-core execution scheme within MUMPS
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
  Direct I/O

Conclusion and Future work
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
  Direct I/O

Conclusion and Future work
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:
- Factors \textit{systematically} written to disk;
- Active Storage kept in memory.

Factors

Active frontal matrix

Stack of contribution blocks

Factors Stack of contribution blocks

Active Storage

Elimination tree

Non-zero Fill-in

A=

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

L+U−I=

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:

- Factors *systematically* written to disk;
- Active Storage kept in memory.

Elimination tree
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:
- Factors *systematically* written to disk;
- Active Storage *kept in memory.*
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:
- Factors *systematically* written to disk;
- Active Storage kept in memory.
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:

- Factors *systematically* written to disk;
- Active Storage kept in memory.

Factors

- Stack of contribution blocks
- Active frontal matrix

Active Storage

Elimination tree
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:

- Factors *systematically* written to disk;
- Active Storage kept in memory.

Factors

<table>
<thead>
<tr>
<th>Active frontal matrix</th>
<th>Stack of contribution blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Elimination tree

Factors

<table>
<thead>
<tr>
<th>Contributions block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

Active Storage
The multifrontal method (Duff, Reid’83)

Storage divided into two parts:

- Factors *systematically* written to disk;
- Active Storage kept in memory.
Memory behaviour (serial postorder traversal)
Memory behaviour (serial postorder traversal)

![Diagram showing a tree with nodes 1, 2, and 3, with node 3 at the top, and node 1 to the left and node 2 to the right. Node 1 has a red section at its bottom right corner.]

Abdou Guermouche, MUMPS Users Group Meeting, April 2010
Memory behaviour (serial postorder traversal)
Memory behaviour (serial postorder traversal)
Memory behaviour (serial postorder traversal)
Memory Behaviour (serial postorder traversal)
Memory Behaviour (serial postorder traversal)
Memory Behaviour (serial postorder traversal)
Parallel multifrontal scheme

- **Type 1**: Nodes processed on a single processor
- **Type 2**: Nodes processed with a parallel 1D blocked factorization
- **Type 3**: Parallel 2D cyclic factorization (root node)
Parallel multifrontal scheme

- **Type 1**: Nodes processed on a single processor
- **Type 2**: Nodes processed with a parallel 1D blocked factorization
- **Type 3**: Parallel 2D cyclic factorization (root node)
Solution step → solve the given system using the factored matrix.

Sequential case:
- forward step (Fwd): postordering as in the factorization phase
- backward step (Bwd): in the reverse order

Parallel case:
- no guarantee of the order in which the nodes are processed
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
  Direct I/O

Conclusion and Future work
What gains can we expect?

Typical memory behavior: Active memory / total memory ratio

![Graph showing the typical memory behavior ratio for different numbers of processors.](image)

### Number of processors
- AUDIKW_1
- CONESHL_MOD
- CONESHL2
- CONV3D
- ULTRASOUND80

10/25
Abdou Guermouche, MUMPS Users Group Meeting, April 2010
Out-of-core storage of factors:

→ write factor to disk as soon as they are computed.
Out-of-core factorization (Phd of E. Agullo)

Out-of-core storage of factors:

→ write factor to disk as soon as they are computed.

Synchronous Version:

- Use standard write operations
- Factors are written to disk as soon as they are computed
Out-of-core storage of factors:

→ write factor to disk as soon as they are computed.

Synchronous Version:

• Use standard write operations
• Factors are written to disk as soon as they are computed

Asynchronous Version:

• Copy factors to a user buffer as soon as they are computed
• A dedicated I/O thread writes factors from the user buffer to disk
Out-of-core factorization (Phd of E. Agullo)

Out-of-core storage of factors:

→ write factor to disk as soon as they are computed.

Synchronous Version:

• Use standard write operations
• Factors are written to disk as soon as they are computed

Asynchronous Version:

• Copy factors to a user buffer as soon as they are computed
• A dedicated I/O thread writes factors from the user buffer to disk

Next step → factors and stack out-of-core (largest problems or many processors)
Performance study: parallel executions (CRAY XD1 system at CERFACS, local disks)

Elapsed time for the factorization step (normalized to the in-core case) - CONESHL_MOD matrix

**RED:** \( \frac{\text{time Asynchronous version}}{\text{time in-core}} \)  **GREEN:** \( \frac{\text{time Synchronous version}}{\text{time in-core}} \)
Volume of I/O minimization

- Assumption: factors written to disk as soon as computed.
- Active memory peak: tree traversal-dependent.

![Diagram showing worst and best case memory peaks]

- **LIU’86**: Optimum algorithm *(MinMEM)* for minimizing the peak of active memory.
- Problem: How to minimize the I/O volume when the active memory does not hold in a given amount of physical memory \( M_0 \)?
Volume of I/O minimization

- Assumption: factors written to disk as soon as computed.
- Active memory peak: tree traversal-dependent.

(c) Worst case
(d) Best case

- LIU’86: Optimum algorithm (MinMEM) for minimizing the peak of active memory.
- Problem: How to minimize the I/O volume when the active memory does not hold in a given amount of physical memory $M_0$?
Proportional mapping VS preorder traversal

Elimination tree:

Mapping

- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks
Proportional mapping:

Mapping

- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks:

- Fine-grain + coarse-grain parallelism;
- Bad memory efficiency.
Proportional mapping VS postorder traversal

Proportional mapping:

512

• Initially: all processors on root node;
• Recursively split the set of processors on child subtrees.

Advantages and drawbacks,
Fine-grain + coarse-grain parallelism;
bad memory efficiency.
Proportional mapping VS postorder traversal

Proportional mapping:

Mapping

• Initially: all processors on root node;
• Recursively split the set of processors on child subtrees.

Advantages and drawbacks

- Fine-grain + coarse-grain parallelism;
- Bad memory efficiency.
Proportional mapping VS postorder traversal

Proportional mapping:

```
256 256
128 128 128 128
512
```

Mapping

- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks

- Fine-grain + coarse-grain parallelism;
- Bad memory efficiency.
Proportional mapping VS postorder traversal

Proportional mapping:

- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks:

- Fine-grain + coarse-grain parallelism;
- Bad memory efficiency.
Proportional mapping VS postorder traversal

Elimination tree:

- Postorder traversal, node by node;
- all processors on each node.

Advantages and drawbacks:

- Only fine-grain parallelism;
- High memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:
- Only fine-grain parallelism;
- High memory efficiency.
Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:
- Only fine-grain parallelism;
- High memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:
- Only fine-grain parallelism;
- High memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:
- Only fine-grain parallelism;
- High memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- all processors on each node.

Advantages and drawbacks:
- Only fine-grain parallelism;
- high memory efficiency.
Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:

😊 Only fine-grain parallelism;
😊 High memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- all processors on each node.

Advantages and drawbacks:

😊 Only fine-grain parallelism;
😊 high memory efficiency.
Proportional mapping VS postorder traversal

Postorder traversal:

- Postorder traversal, node by node;
- All processors on each node.

Advantages and drawbacks:

- Only fine-grain parallelism;
- High memory efficiency.
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
  Direct I/O

Conclusion and Future work
Assumptions:

- During factorization all factors are written to local disks
- No factors are kept in memory at the beginning of the solution step

How to load efficiently data from disk?

- Each factor-block is loaded only once
- User control of number and size of buffers
- One Emergency buffer (EMG), to hold largest front (demand driven)
- Other buffers used to automatically prefetch data with a look-ahead mechanism
Out-of-core Solution step (Phd of T. Slavova)

Assumptions:

- During factorization all factors are written to local disks
- No factors are kept in memory at the beginning of the solution step

How to load efficiently data from disk?

- Each factor-block is loaded only once
- User control of number and size of buffers
- One Emergency buffer (EMG), to hold largest front (demand driven)
- Other buffers used to automatically prefetch data with a look-ahead mechanism
Assumptions:

- During factorization all factors are written to local disks
- No factors are kept in memory at the beginning of the solution step

How to load efficiently data from disk?

- Each factor-block is loaded only once
- User control of number and size of buffers
- One Emergency buffer (EMG), to hold largest front (demand driven)
- Other buffers used to automatically prefetch data with a look-ahead mechanism
Out-of-core Solution step (Phd of T. Slavova)

Assumptions:

- During factorization all factors are written to local disks
- No factors are kept in memory at the beginning of the solution step

How to load efficiently data from disk?

- Each factor-block is loaded only once
- User control of number and size of buffers
- One Emergency buffer (EMG), to hold largest front (demand driven)
- Other buffers used to automatically prefetch data with a look-ahead mechanism
Scheduling for the solution step

**Pool of tasks:** list of all tasks ready to be executed (scheduling)

- **Illustration:** sequential processing of the tree

Pool at the beginning of FWD

I - II step

| 10 | 7 | 6 | 4 | 2 | 1 |

III step

| 10 | 7 | 6 | 4 | 3 |

Factors Data on the HARD DISK

<table>
<thead>
<tr>
<th>FWD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1   2   3   4   5   6   7   8   9   10  11</td>
</tr>
</tbody>
</table>

Factors Data on the HARD DISK
Scheduling for the solution step

**Pool of tasks:** list of all tasks ready to be executed (scheduling)

• *Illustration: sequential processing of the tree*

Pool at the beginning of FWD

I - II step

```
10 7 6 4 2 1
```

III step

```
10 7 6 4 3
```

Pool at the beginning of BWD

I step

```
11
```

II step

```
9 10
```

Factors Data on the HARD DISK

```
1 2 3 4 5 6 7 8 9 10 11
```

FWD

```
1 2 3 4 5 6 7 8 9 10 11
```

BWD

```
11 10 9
```

Abdou Guermouche, MUMPS Users Group Meeting, April 2010
### Experimental results

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Nb of Procs</th>
<th>Factor Size (MB)</th>
<th>Workspace (MB)</th>
<th>Fwd (sec)</th>
<th>Bwd (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFO</td>
<td>1</td>
<td>2534</td>
<td>12</td>
<td>171.5</td>
<td>177.2</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>170.6</td>
<td>176.8</td>
</tr>
<tr>
<td>LIFO</td>
<td>8</td>
<td>317</td>
<td>12</td>
<td>25.2</td>
<td>137.6</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>29.0</td>
<td>45.2</td>
</tr>
<tr>
<td>LIFO</td>
<td>32</td>
<td>79</td>
<td>8.2</td>
<td>11.0</td>
<td>53.1</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>10.2</td>
<td>10.7</td>
</tr>
</tbody>
</table>
## Experimental results

### AMANDE (CEA-CESTA)

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Nb of Procs</th>
<th>Factor Size (MB)</th>
<th>Workspace (MB)</th>
<th>Fwd (sec)</th>
<th>Bwd (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFO</td>
<td>20</td>
<td>1625</td>
<td>425</td>
<td>725.9</td>
<td>964.8</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>678.0</td>
<td>866.1</td>
</tr>
<tr>
<td>LIFO</td>
<td>24</td>
<td>1364</td>
<td>366</td>
<td>679.8</td>
<td>1071.6</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>475.5</td>
<td>629.5</td>
</tr>
<tr>
<td>LIFO</td>
<td>32</td>
<td>1028</td>
<td>261</td>
<td>358.9</td>
<td>814.6</td>
</tr>
<tr>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td>350.9</td>
<td>564.6</td>
</tr>
</tbody>
</table>
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
   Direct I/O

Conclusion and Future work
I/O Mechanisms

read and write operations use a cache mechanism (page cache)

- For each call to read or write, data is kept in the page cache at the kernel level
- User doesn’t know when data is “really” written to disk (unless by explicit synchronization)
- User has no control on the size of the page cache
- The page cache is usually managed with a LRU scheme
I/O Mechanisms

read and write operations use a cache mechanism (page cache)

• For each call to read or write, data is kept in the page cache at the kernel level
• User doesn’t know when data is “really” written to disk (unless by explicit synchronization)
• User has no control on the size of the page cache
• The page cache is usually managed with a LRU scheme

In our context, page cache can be dangerous.

• I/O may not have the same speed (depending on whether disk is accessed or not)
• The kernel may dramatically slowdown the performance of I/O’s.
I/O Mechanisms

read and write operations use a cache mechanism (page cache)

• For each call to read or write, data is kept in the page cache at the kernel level
• User doesn’t know when data is “really” written to disk (unless by explicit synchronization)
• User has no control on the size of the page cache
• The page cache is usually managed with a LRU scheme

In our context, page cache can be dangerous.

• I/O may not have the same speed (depending on whether disk is accessed or not)
• The kernel may dramatically slowdown the performance of I/O’s.

⇒ Use of direct I/O mechanisms
Direct I/O scheme

Advantages:
- Data is directly written to disk (data is not copied in the page cache)
- Very efficient I/O operations

Drawbacks:
- A disk access is made at each call to read or write
- Data needs to be aligned in memory

Direct I/O scheme ⇒ Use of more sophisticated algorithms but ensures robustness.
Direct I/O scheme

Advantages:
- Data is directly written to disk (data is not copied in the page cache)
- Very efficient I/O operations

Drawbacks:
- A disk access is made at each call to read or write
- Data needs to be aligned in memory

Direct I/O scheme ⇒ Use of more sophisticated algorithms but ensures robustness.

Preliminary results: Factorization time (seconds)

<table>
<thead>
<tr>
<th></th>
<th>Direct I/O Sync.</th>
<th>Direct I/O Async.</th>
<th>P.C. Sync.</th>
<th>P.C. Async.</th>
<th>in-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDIKW_1</td>
<td>2417.1</td>
<td>2217.3</td>
<td>2260.8</td>
<td>2211.3</td>
<td>2126.4</td>
</tr>
<tr>
<td>CONESHL_MOD</td>
<td>995.6</td>
<td>967.2</td>
<td>979.2</td>
<td>953.6</td>
<td>930.4</td>
</tr>
<tr>
<td>CONV3D64</td>
<td>10826.9</td>
<td>7599.4</td>
<td>8078.4</td>
<td>7981.6</td>
<td>-</td>
</tr>
<tr>
<td>ULTRASOUND80</td>
<td>1446.9</td>
<td>1389.8</td>
<td>1436.4</td>
<td>1377.3</td>
<td>1382.5</td>
</tr>
</tbody>
</table>
Direct I/O scheme

Advantages:
• Data is directly written to disk (data is not copied in the page cache)
• Very efficient I/O operations

Drawbacks:
• A disk access is made at each call to read or write
• Data needs to be aligned in memory

Direct I/O scheme $\Rightarrow$ Use of more sophisticated algorithms but ensures robustness.

Preliminary results: Factorization time (seconds)

<table>
<thead>
<tr>
<th></th>
<th>Direct I/O</th>
<th>Direct I/O</th>
<th>P.C.</th>
<th>P.C.</th>
<th>in-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDIKW_1</td>
<td>2417.1</td>
<td>2217.3</td>
<td>2260.8</td>
<td>2211.3</td>
<td>2126.4</td>
</tr>
<tr>
<td>CONESHL_MOD</td>
<td>995.6</td>
<td>967.2</td>
<td>979.2</td>
<td>953.6</td>
<td>930.4</td>
</tr>
<tr>
<td>CONV3D64</td>
<td>10826.9</td>
<td>7599.4</td>
<td>8078.4</td>
<td>7981.6</td>
<td>-</td>
</tr>
<tr>
<td>ULTRASOUND80</td>
<td>1446.9</td>
<td>1389.8</td>
<td>1436.4</td>
<td>1377.3</td>
<td>1382.5</td>
</tr>
</tbody>
</table>
Direct I/O scheme

Advantages:
- Data is directly written to disk (data is not copied in the page cache)
- Very efficient I/O operations

Drawbacks:
- A disk access is made at each call to read or write
- Data needs to be aligned in memory

Direct I/O scheme ⇒ Use of more sophisticated algorithms but ensures robustness.

Preliminary results: Time for solution step (Qimonda07 matrix)

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct I/O (Demand-driven)</td>
<td>1149.2</td>
<td>1279.2</td>
</tr>
<tr>
<td>Direct I/O (Look-ahead)</td>
<td>174.0</td>
<td>183.7</td>
</tr>
<tr>
<td>P.C. (Demand-driven)</td>
<td>186.4</td>
<td>207.7</td>
</tr>
</tbody>
</table>
Outline

Multifrontal method

out-of-core factorization step

out-of-core solution step

Operating system I/O mechanisms
  Direct I/O

Conclusion and Future work
Conclusion

• Implementation of an out-of-core extension of MUMPS.
  ◦ Available to the community since two years.
  ◦ 2 PhD thesis in this context.
  ◦ Everything has not been made available to the users yet.

• What still has to be integrated?
  ◦ Direct I/O scheme.
  ◦ I/O driven scheduling for solution step.
  ◦ ...

• Out-of-core related features.
  ◦ 64-bit addressing for internal arrays.
  ◦ Communication buffer size reduction.
  ◦ Interleaved I/O operations (with computations) for the processing of frontal matrices.
Future work

- Design and study memory scalable algorithms with a good performance behaviour.
  ➜ In the continuation of Emmanuel’s thesis.
  ➜ François-Henri will work on this topic.
- Improve the out-of-core API.
- Do we need to go further? (we hope so)
  - Out-of-core dynamic memory management.
  - Integration of the I/O minimizing algorithms (and their adaptation to the parallel context).
  - ...