out-of-core extension of the MUMPS solver

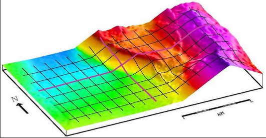
Abdou Guermouche, LaBRI Bordeaux

MUMPS Users Group Meeting, April 2010

MUMPS

Context

Solving sparse linear systems



$$Ax = b$$

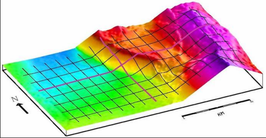
$\Rightarrow$ Direct methods: $A = LU$

Typical matrix: BRGM matrix

- 3.7×10^6 variables
- 156×10^6 non zeros in A
- 4.5×10^9 non zeros in LU
- 26.5×10^{12} flops

Context

Solving sparse linear systems



$$Ax = b$$

$\Rightarrow$ Direct methods: $A = LU$

Typical matrix: BRGM matrix

- 3.7×10^6 variables
- 156×10^6 non zeros in A
- 4.5×10^9 non zeros in LU
- 26.5×10^{12} flops

Context

Physical constraint

Core memory

Memory required

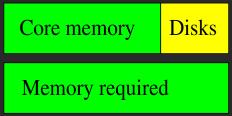
Memory crash

Software challenge

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

Context

Out-of-core



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

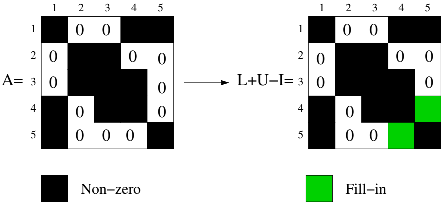
out-of-core solution step

Operating system I/O mechanisms

- Direct I/O

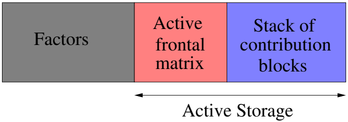
Conclusion and Future work

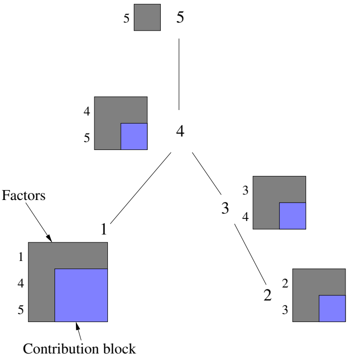
The multifrontal method (Duff, Reid'83)



Storage divided into two parts:

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



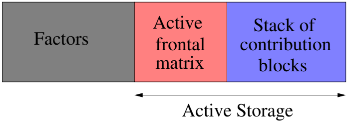


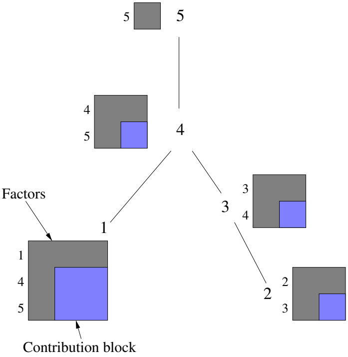
Elimination tree

The multifrontal method (Duff, Reid'83)

Storage divided into two parts:

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



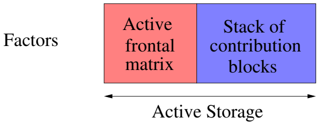


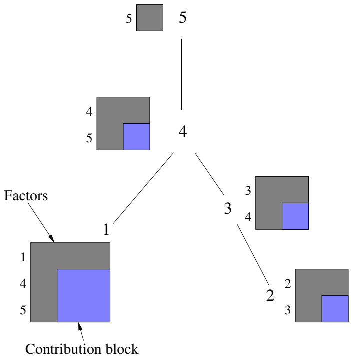
Elimination tree

The multifrontal method (Duff, Reid'83)

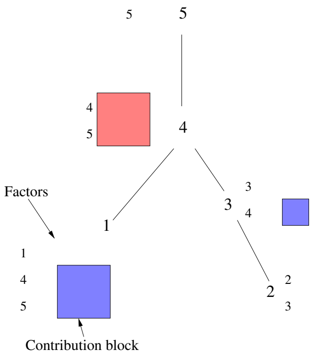
Storage divided into two parts:

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



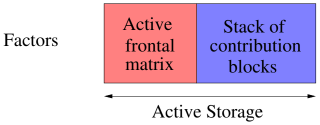


Elimination tree



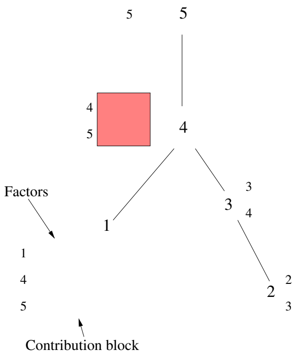
Storage divided into two parts:

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



Elimination tree

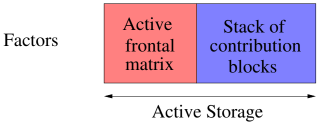
The multifrontal method (Duff, Reid'83)



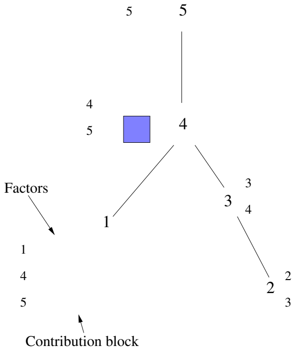
Elimination tree

Storage divided into two parts:

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



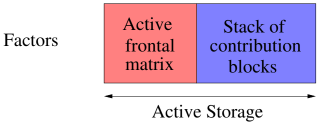
The multifrontal method (Duff, Reid'83)



Elimination tree

Storage divided into two parts:

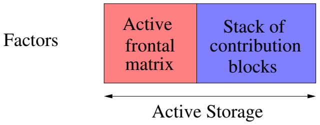
- Factors *systematically* written to disk;
- Active Storage kept in memory.

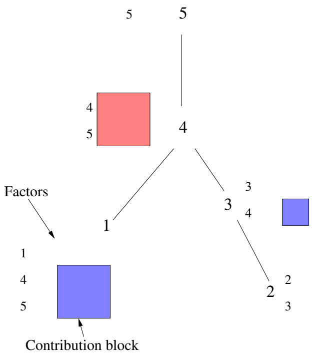


The multifrontal method (Duff, Reid'83)

Storage divided into two parts:

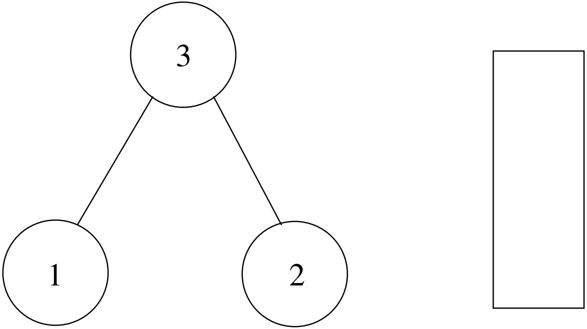
- Factors *systematically* written to disk;
- Active Storage kept in memory.



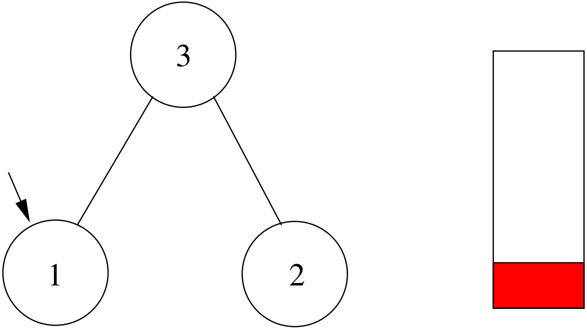


Elimination tree

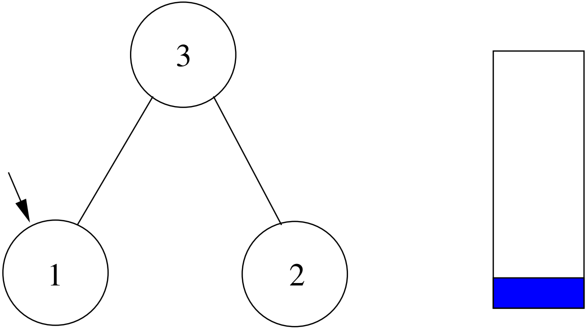
Memory behaviour (serial postorder traversal)



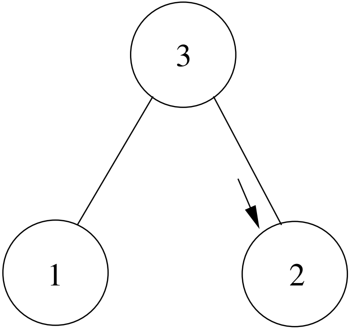
Memory behaviour (serial postorder traversal)

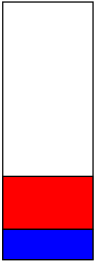


Memory behaviour (serial postorder traversal)

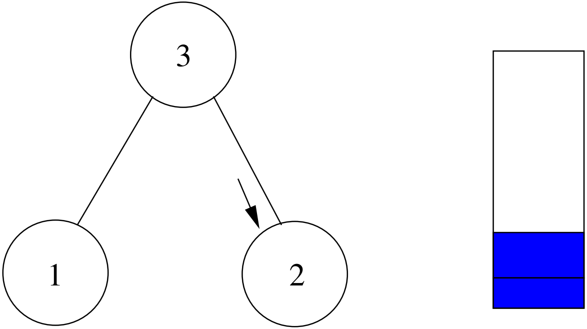


Memory behaviour (serial postorder traversal)

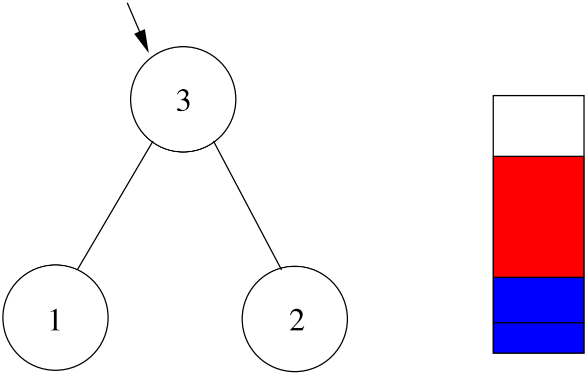




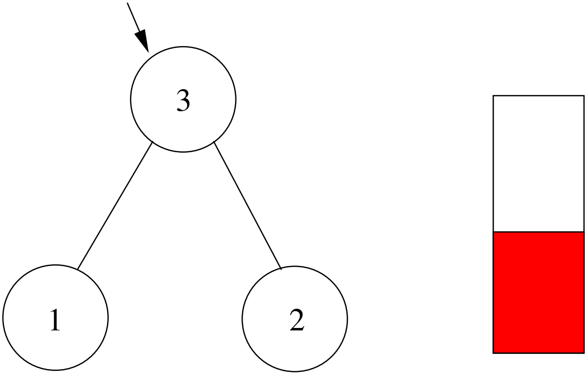
Memory Behaviour (serial postorder traversal)



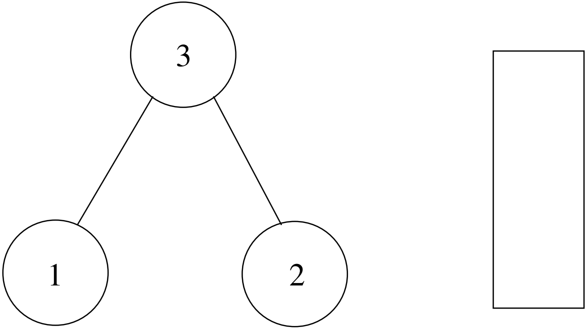
Memory behaviour (serial postorder traversal)



Memory behaviour (serial postorder traversal)

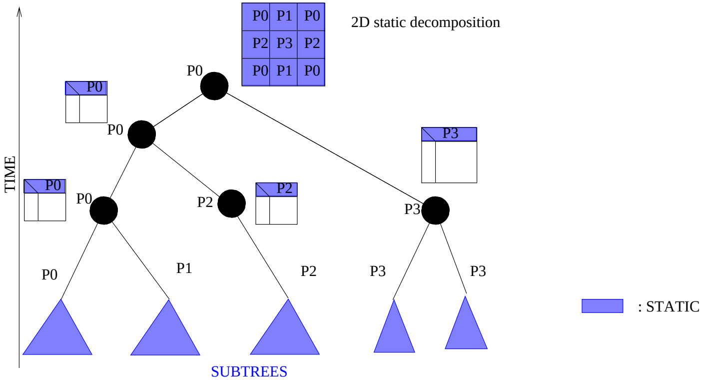


Memory behaviour (serial postorder traversal)



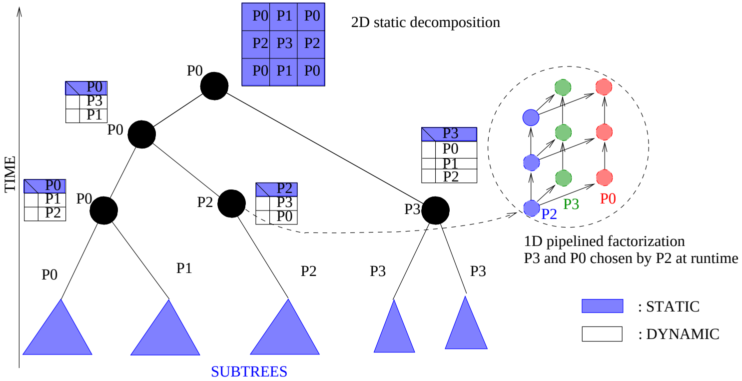
Parallel multifrontal scheme

- 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)



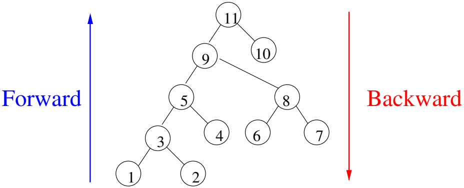
Parallel multifrontal scheme

- 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

Solution step → solve the given system using the factored matrix.



Assembly Tree

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

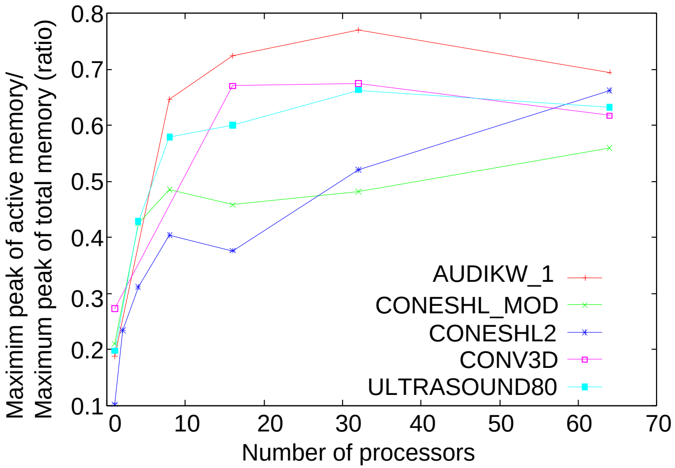
out-of-core solution step

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



Out-of-core factorization (Phd of E. Agullo)

Out-of-core storage of factors :

$\longrightarrow$ write factor to disk as soon as they are computed.

Out-of-core factorization (Phd of E. Agullo)

Out-of-core storage of factors :

$\rightarrow$ 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 factorization (Phd of E. Agullo)

Out-of-core storage of factors :

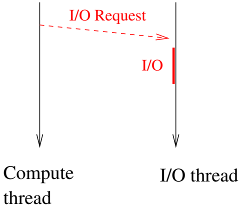
$\longrightarrow$ write factor to disk as soon as they are computed.

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 :

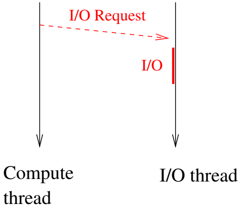
→ write factor to disk as soon as they are computed.

Synchronous Version:

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

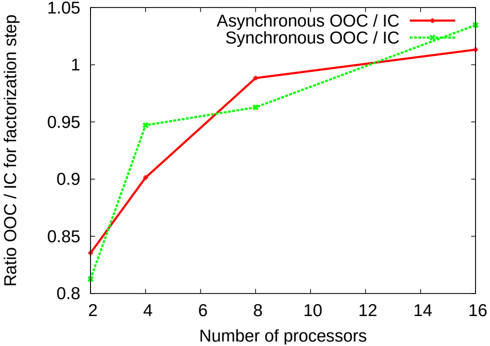
Asynchronous Version:

- 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)

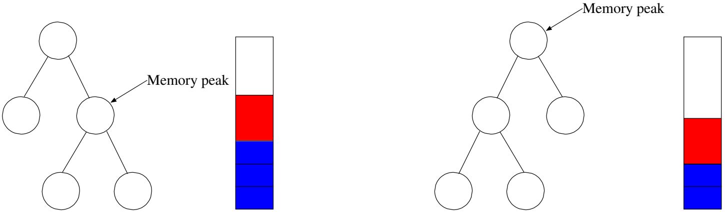


Elapsed time for the factorization step (normalized to the in-core case) - CONESHL_MOD matrix

RED: $\frac{\textit{time Asynchronous version}}{\textit{time in-core}}$ GREEN: $\frac{\textit{time Synchronous version}}{\textit{time in-core}}$

Volume of I/O minimization

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

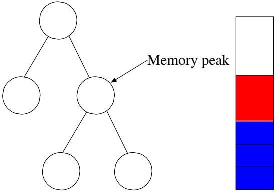


(a) Worst case (b) Best case

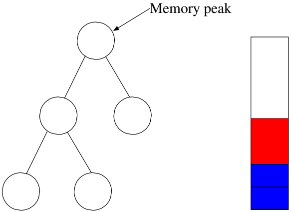
- Liu'86: Optimum algorithm (MinMEM) for minimizing the peak of active memory.
- Problem: How to minimize the I/O volume when the active memory does not hold in a given amount of physical memory M_0 ?

Volume of I/O minimization

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



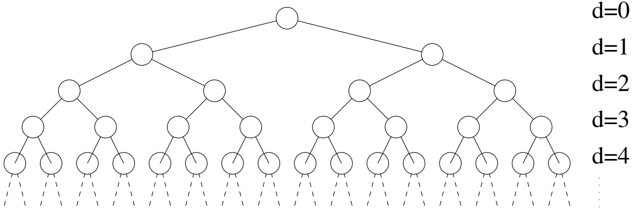
(c) Worst case



(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 ?

Elimination tree :



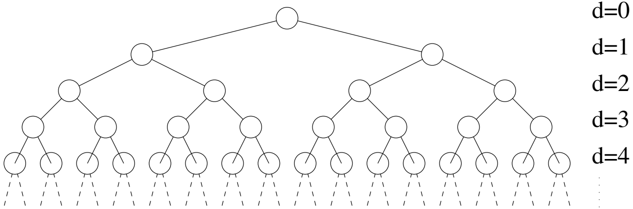
Mapping

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Proportional mapping:



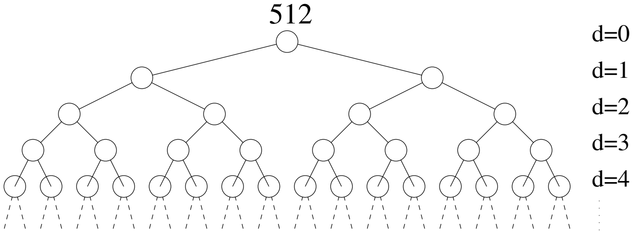
Mapping

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Proportional mapping:



Mapping

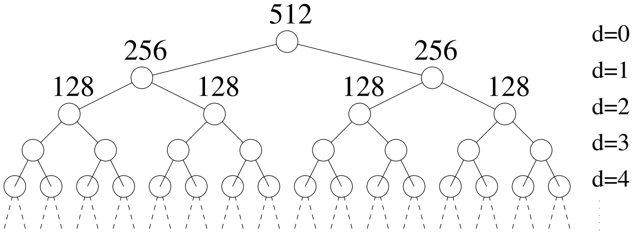
- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks

- ☺ Fine-grain + coarse-grain parallelism;
- ☹ bad memory efficiency;

Proportional mapping VS postorder traversal

Proportional mapping:



Mapping

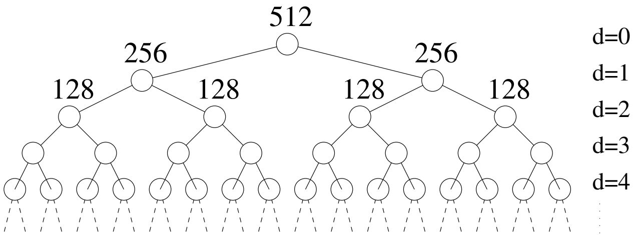
- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks

- ☺ Fine-grain + coarse-grain parallelism;
- ☹ bad memory efficiency.

Proportional mapping VS postorder traversal

Proportional mapping:



Mapping

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

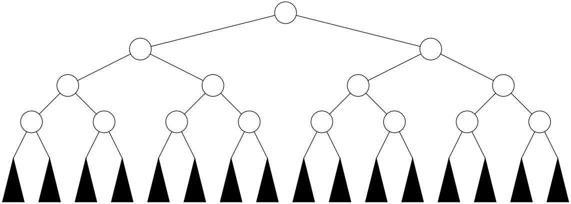
Advantages and drawbacks

☺ Fine-grain + coarse-grain parallelism;

☹ bad memory efficiency.

Proportional mapping VS postorder traversal

Proportional mapping:



Mapping

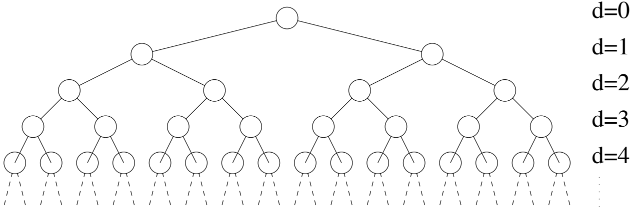
- Initially: all processors on root node;
- Recursively split the set of processors on child subtrees.

Advantages and drawbacks

- ☺ Fine-grain + coarse-grain parallelism;
- ☹ bad memory efficiency.

Proportional mapping VS postorder traversal

Elimination tree :



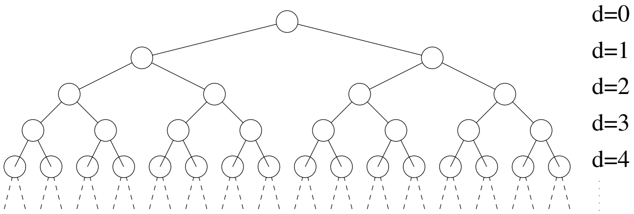
Traversal

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Postorder traversal :



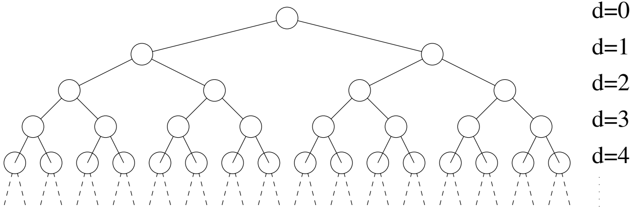
Traversal

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Postorder traversal :



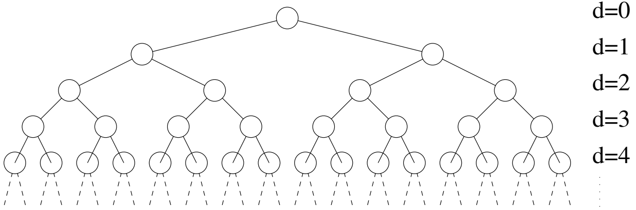
Traversal

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Postorder traversal :

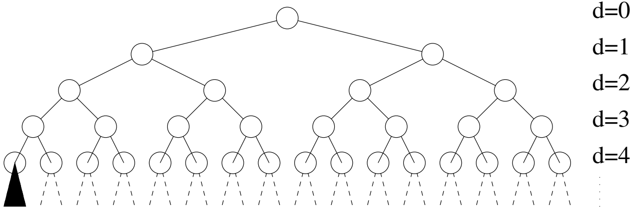


Traversal

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

Proportional mapping VS postorder traversal

Postorder traversal :



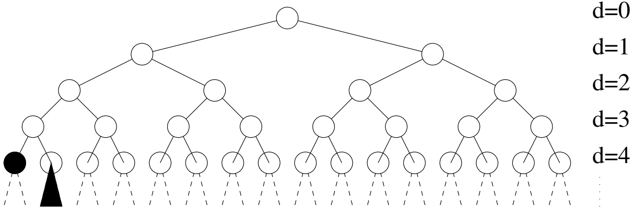
Traversal

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

Advantages and drawbacks

Proportional mapping VS postorder traversal

Postorder traversal :



Traversal

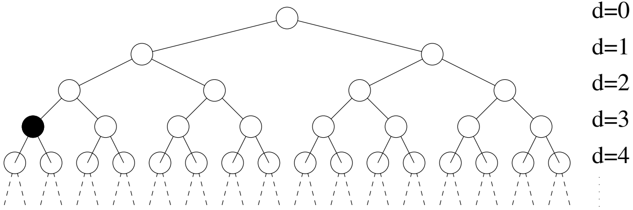
- Postorder traversal, node by node;
- all processors on each node.

Advantages and drawbacks

high memory efficiency

Proportional mapping VS postorder traversal

Postorder traversal :



Traversal

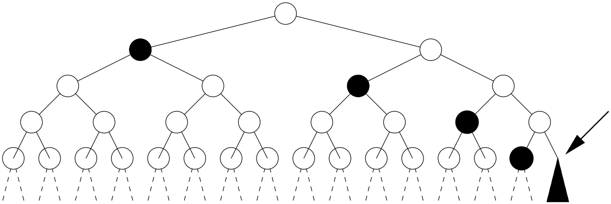
- Postorder traversal, node by node;
- all processors on each node.

Advantages and drawbacks

- ☹ Only fine-grain parallelism,
- ☺ high memory efficiency.

Proportional mapping VS postorder traversal

Postorder traversal :



Traversal

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

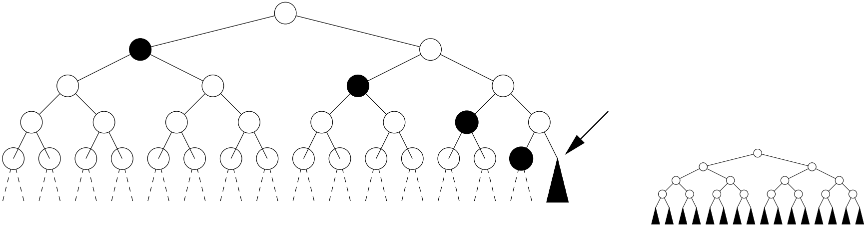
Advantages and drawbacks

☹ Only fine-grain parallelism;

☺ high memory efficiency.

Proportional mapping VS postorder traversal

Postorder traversal :



Traversal

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

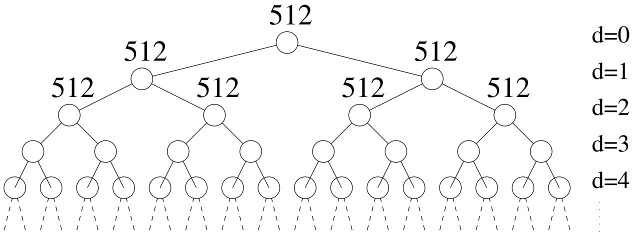
Advantages and drawbacks

☹ Only fine-grain parallelism;

☺ high memory efficiency.

Proportional mapping VS postorder traversal

Postorder traversal :



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

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

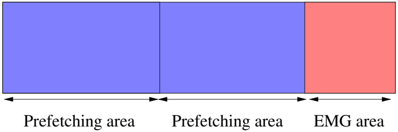
Out-of-core Solution step (Phd of T. Slavova)

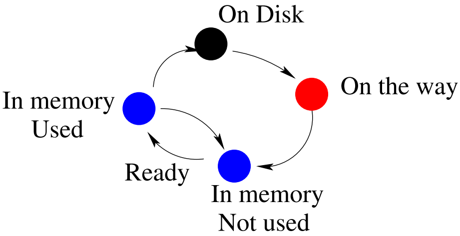
Assumptions:

- 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





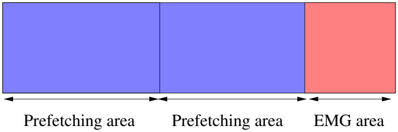
Out-of-core Solution step (Phd of T. Slavova)

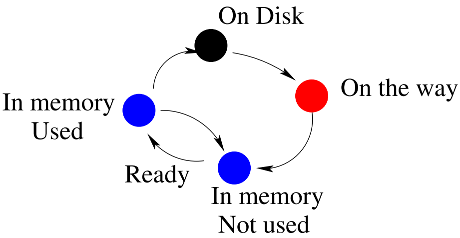
Assumptions:

- 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





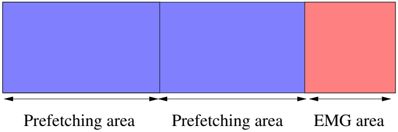
Out-of-core Solution step (Phd of T. Slavova)

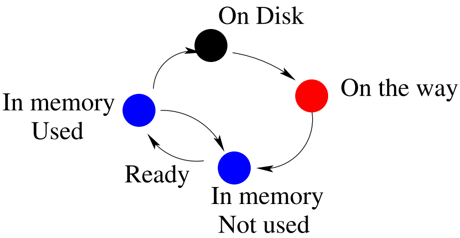
Assumptions:

- 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

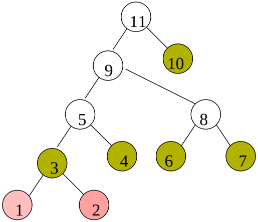




Scheduling for the solution step

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

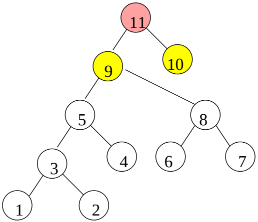
FWD

1	2	3	4	5	6	7	8	9	10	11

Factors Data on the HARD DISK

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	

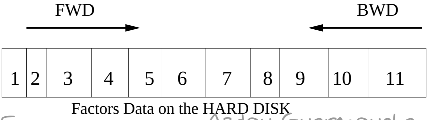
Pool at the beginning of BWD

I step

11						

II step

9	10					



Experimental results

QIMONDA07 (Qimonda AG company)

Strategy	Nb of Procs	Factor Size (MB)	Workspace (MB)	Fwd (sec)	Bwd (sec)
LIFO				171.5	177.2
NNS	1	2 534	12	170.6	176.8
LIFO				25.2	137.6
NNS	8	317	12	29.0	45.2
LIFO				11.0	53.1
NNS	32	79	8.2	10.2	10.7

Experimental results

AMANDE (CEA-CESTA)

Strategy	Nb of Procs	Factor Size (MB)	Workspace (MB)	Fwd (sec)	Bwd (sec)
LIFO				725.9	964.8
NNS	20	1625	425	678.0	866.1
LIFO				679.8	1071.6
NNS	24	1364	366	475.5	629.5
LIFO				358.9	814.6
NNS	32	1028	261	350.9	564.6

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

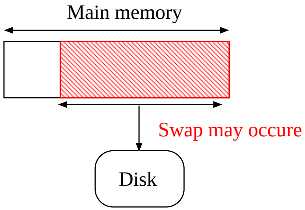
I/O Mechanisms

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.



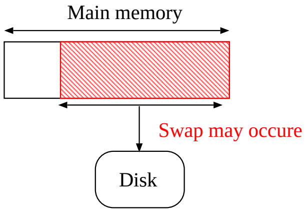
I/O Mechanisms

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.



$\Rightarrow$ 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 $\Rightarrow$ 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)

	Direct I/O Sync.	Direct I/O Async.	P.C. Sync.	P.C. Async.	*in-core*
AUDIKW_1	2417.1	2217.3	2260.8	2211.3	2126.4
CONESHL_MOD	995.6	967.2	979.2	953.6	930.4
CONV3D64	10826.9	7599.4	8078.4	7981.6	-
ULTRASOUND80	1446.9	1389.8	1436.4	1377.3	1382.5

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)

	Direct I/O Sync.	Direct I/O Async.	P.C. Sync.	P.C. Async.	*in-core*
AUDIKW_1	2417.1	2217.3	2260.8	2211.3	2126.4
CONESHL_MOD	995.6	967.2	979.2	953.6	930.4
CONV3D64	10826.9	7599.4	8078.4	7981.6	-
ULTRASOUND80	1446.9	1389.8	1436.4	1377.3	1382.5

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: Time for solution step (`Qimonda07` matrix)

	Forward	Backward
Direct I/O (Demand-driven)	1149.2	1279.2
Direct I/O (Look-ahead)	174.0	183.7
P.C. (Demand-driven)	186.4	207.7

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).
  - ...