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

Introduction

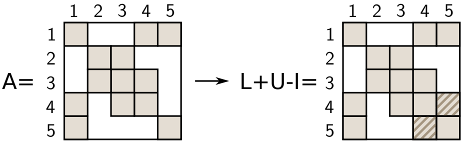
Motivation

- 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

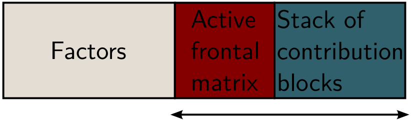
- We want to design mapping algorithms that enforce some memory constraints and provide better memory estimates.
- We focus on multifrontal methods (e.g., MUMPS).

The multifrontal method [Duff & Reid '83]

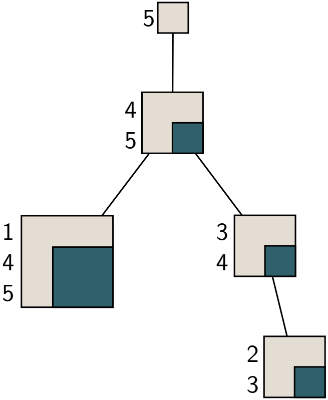


Storage is divided into two parts:

- Factors
- Active memory

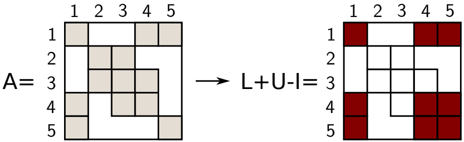


Active storage



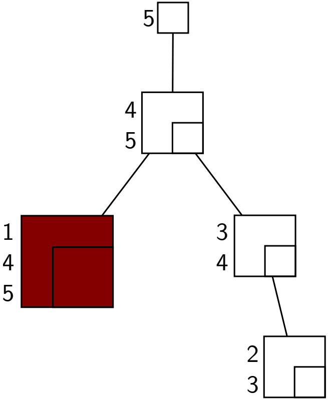
Elimination tree

The multifrontal method [Duff & Reid '83]

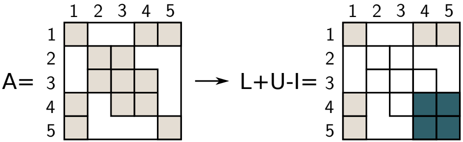


Storage is divided into two parts:

- Factors
- Active memory

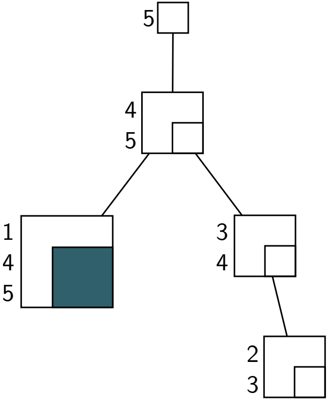


Active memory



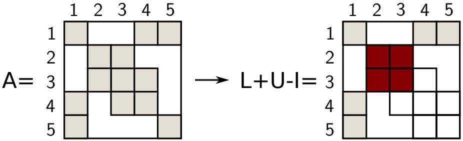
Storage is divided into two parts:

- Factors
- Active memory



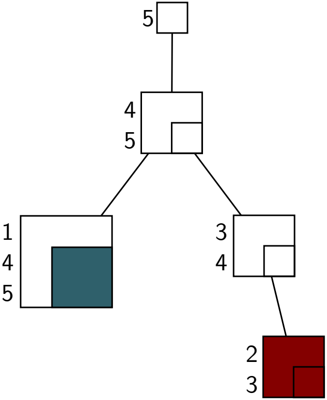
Active memory

The multifrontal method [Duff & Reid '83]



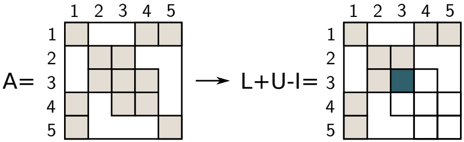
Storage is divided into two parts:

- Factors
- Active memory



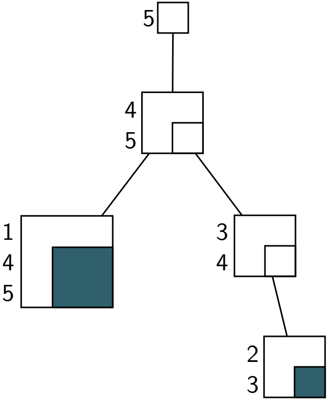
Active memory

The multifrontal method [Duff & Reid '83]

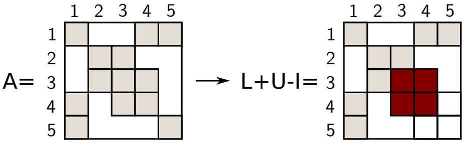


Storage is divided into two parts:

- Factors
- Active memory

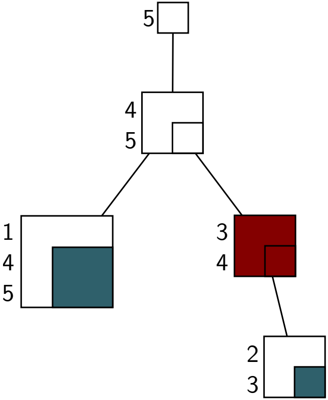


Active memory



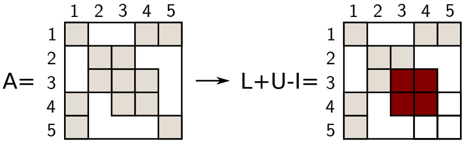
Storage is divided into two parts:

- Factors
- Active memory



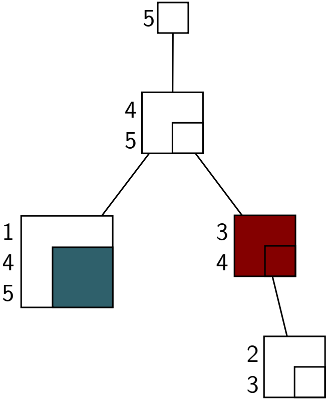
Active memory

The multifrontal method [Duff & Reid '83]



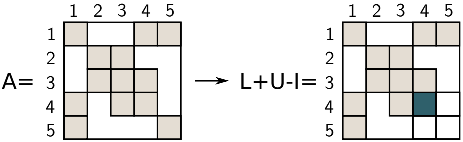
Storage is divided into two parts:

- Factors
- Active memory



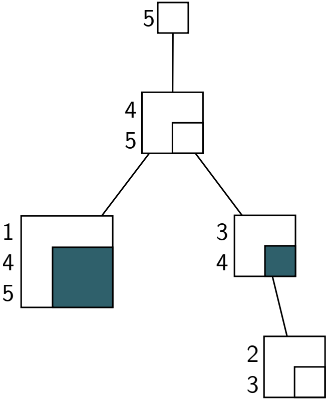
Active memory

The multifrontal method [Duff & Reid '83]



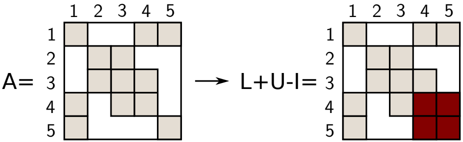
Storage is divided into two parts:

- Factors
- Active memory



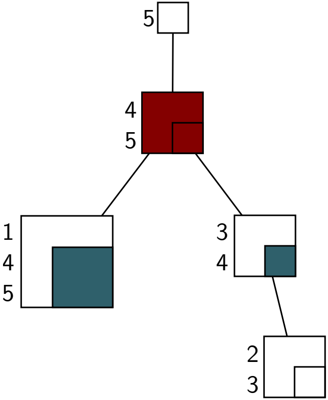
Active memory

The multifrontal method [Duff & Reid ’83]

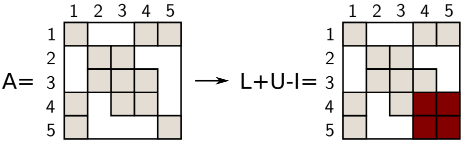


Storage is divided into two parts:

- Factors
- Active memory

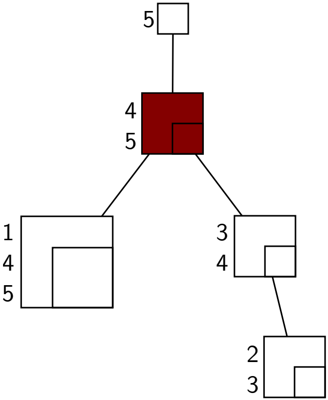


Active memory



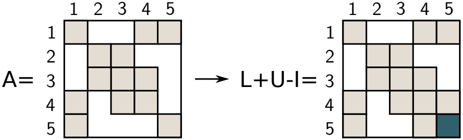
Storage is divided into two parts:

- Factors
- Active memory



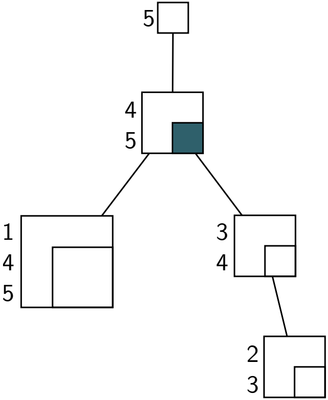
Active memory

The multifrontal method [Duff & Reid '83]

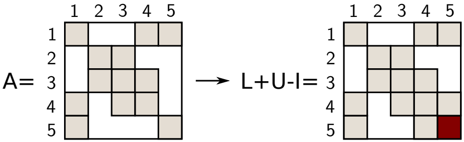


Storage is divided into two parts:

- Factors
- Active memory

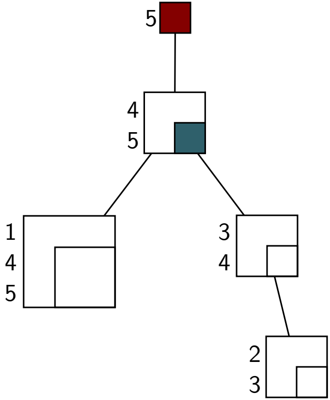


Active memory

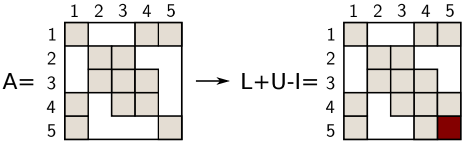


Storage is divided into two parts:

- Factors
- Active memory

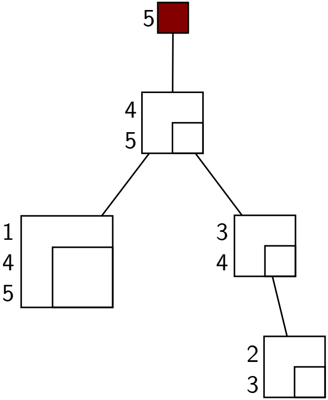


Active memory



Storage is divided into two parts:

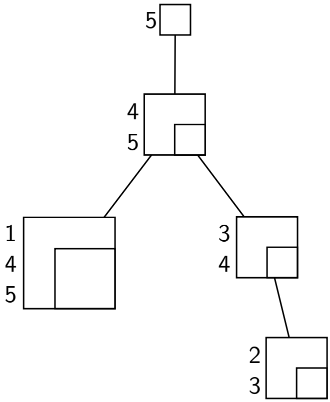
- Factors
- Active memory



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%



Active memory

The problem

Metric: memory efficiency

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

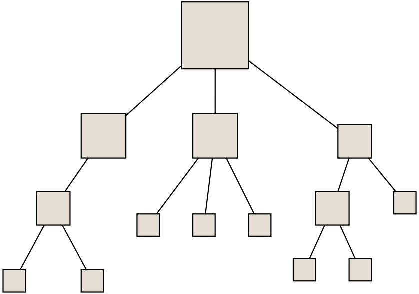
We would like $e(p) \simeq 1$, i.e. S_{seq}/p on each processor.

Common mappings/schedulings provide a poor memory efficiency:

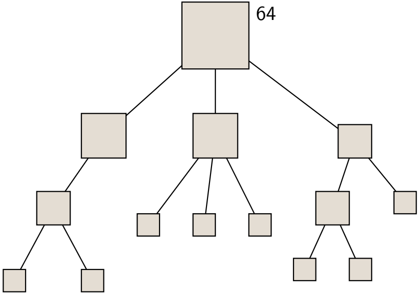
- Proportional mapping: $\lim_{p \to \infty} e(p) = 0$ on regular problems.
- MUMPS relies primarily on a relaxed proportional mapping; typical efficiency: between 0.10 and 0.40.
  Memory estimates are unreliable.

Proportional mapping

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.

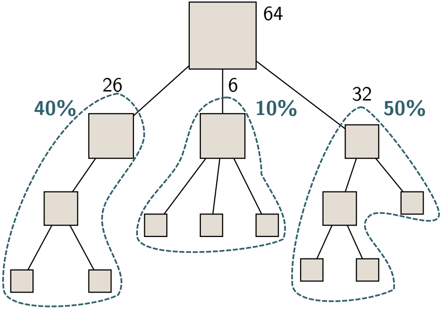


Proportional mapping: top-down traversal of the tree, where processors assigned to a node are distributed among its children proportionally to the weight of their respective subtrees.



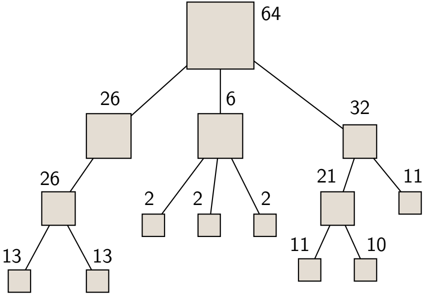
Proportional mapping

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

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.

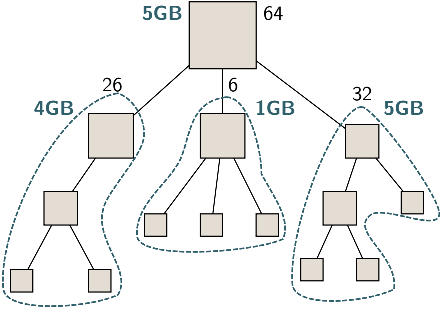


- Targets run time but poor memory efficiency.
- Usually a relaxed version is used: more memory-friendly but unreliable estimates.

Proportional mapping

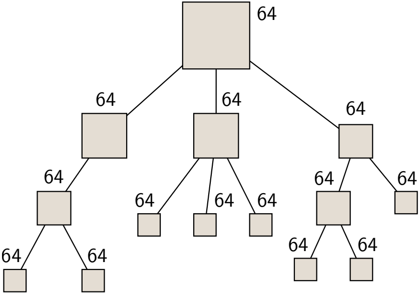
Proportional mapping: assuming that the sequential peak is 5 GB,

$$S_{max}(p) \geqslant \max\left\{\frac{4\,\text{GB}}{26}, \frac{1\,\text{GB}}{6}, \frac{5\,\text{GB}}{32}\right\} = 0.16\ \text{GB} \Rightarrow e(p) \leqslant \frac{5}{64 \times 0.16} \leqslant 0.5$$



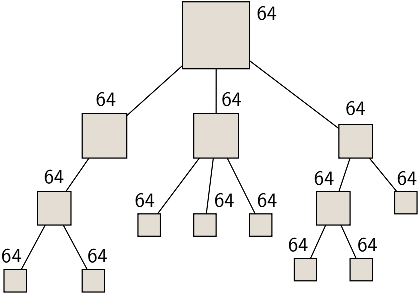
A more memory-friendly strategy...

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



A more memory-friendly strategy...

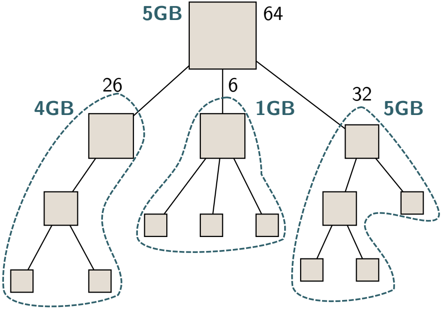
All-to-all mapping: postorder traversal of the tree, where all the processors work at every node:



Optimal memory scalability ($S_{max}(p) = S_{seq}/p$) but no tree parallelism and prohibitive amounts of communications.

A class of "memory-aware" mappings

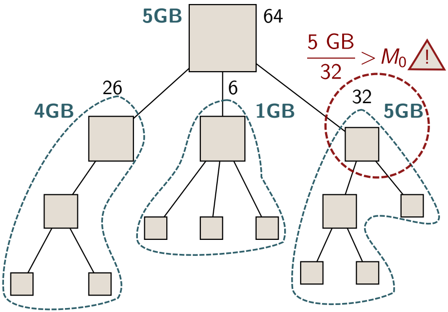
"Memory-aware" mapping [Agullo '08]: aims at enforcing a given memory constraint (M_0, maximum memory per processor):



1. Try to apply proportional mapping.

A class of “memory-aware” mappings

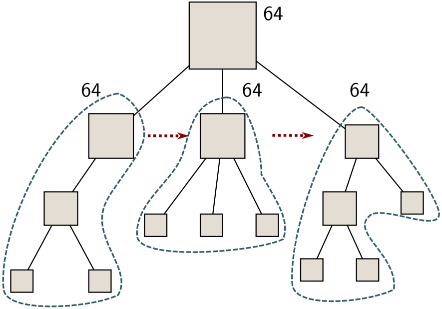
“Memory-aware” mapping [Agullo ’08]: aims at enforcing a given memory constraint (M_0, maximum memory per processor):



1. Try to apply proportional mapping.
2. Enough memory for each subtree?

A class of "memory-aware" mappings

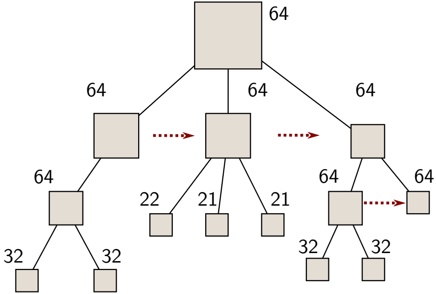
"Memory-aware" mapping [Agullo '08]: aims at enforcing a given memory constraint (M_0, maximum memory per processor):



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.

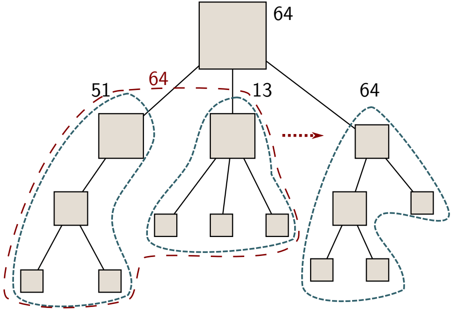
A class of "memory-aware" mappings

"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
  $\Longrightarrow$ performance issues on parallel nodes.

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



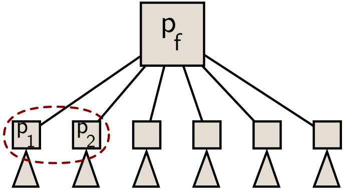
Find groups on which proportional mapping works, and serialize these groups. Heuristic: follow a given order (e.g. the serial postorder) and form groups as large as possible.

Memory-aware with groups

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



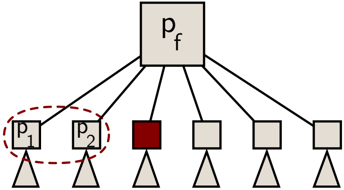
Memory-aware with groups

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

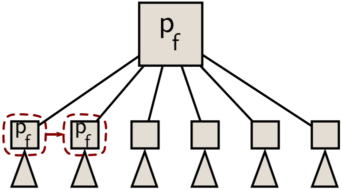


The algorithm maps each child i on p_i processors so that, on every processor working on i, two conditions are enforced:

(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



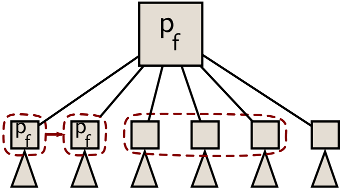
Memory-aware with groups

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



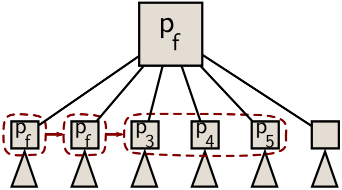
Memory-aware with groups

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.
  $N = 7$ M, $nnz = 189$ 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 GB/256=180MB.

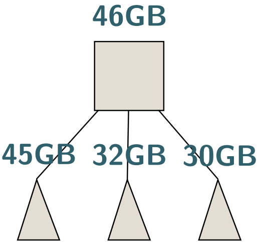
Experiments

	Prop Map	Memory-aware mapping		
		$M_0 = 225$ MB	$M_0 = 380$ MB	
			w/o groups	w/ groups
Max stack peak (MB)	1932	227	330	381
Avg stack peak (MB)	626	122	177	228
Time (s)	1323	2690	2079	1779

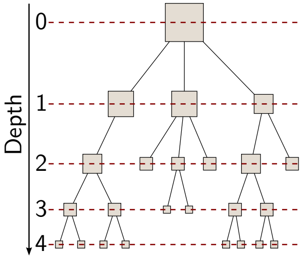
The root node has 3 children that cannot be mapped using a proportional mapping:

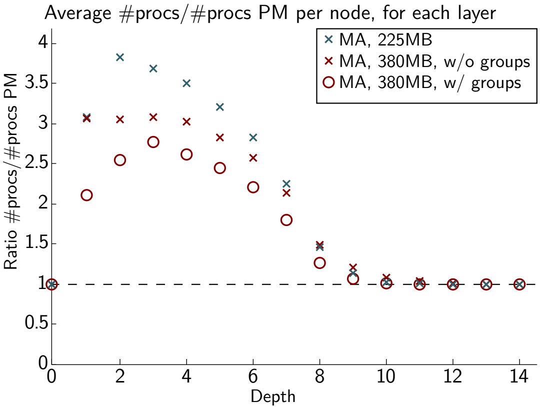
$$46 \text{ GB} + 32 \text{ GB} + 30 \text{ GB} > 380 \text{ MB} \times 256$$

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



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

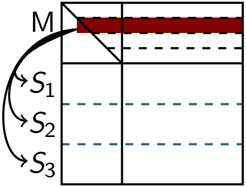




- Relaxing M_0 enhances tree parallelism.
- The variant with groups enhances tree parallelism.

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.

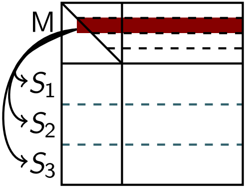


(unsymmetric case)

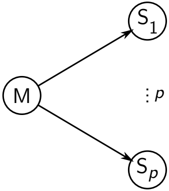
Non-blocking broadcast of a block of factors from the master process.
A kind of ibcast.

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.



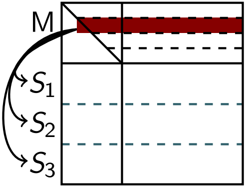
(unsymmetric case)



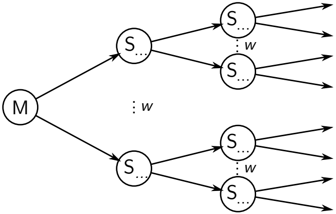
Baseline implementation: a loop of non-blocking sends from the master to the processors. Communication speed is constant.
Hyperion (CICT): 1.7 GB/s; Hopper (NERSC): 4.5 GB/s.

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.



(unsymmetric case)



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.

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

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

BLAS

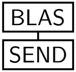
Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full **then**
  Call **Main loop**
end if
Free stuff

Task 1

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

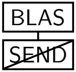


Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full **then**
  Call **Main loop**
end if
Free stuff

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

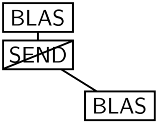


Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full **then**
  Call **Main loop**
end if
Free stuff

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.

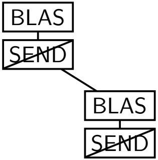
Main loop:
while . . . **do**
  Probe for message
  if message received **then**
    Read tag, execute associated task
      (e.g., **Facto panel)**
  end if
end while



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

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.



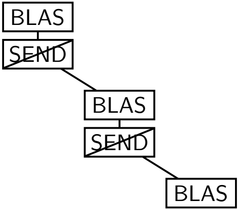
Main loop:
while . . . **do**
  Probe for message
  if message received **then**
    Read tag, execute associated task
        (e.g., **Facto panel)**
  end if
end while

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

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

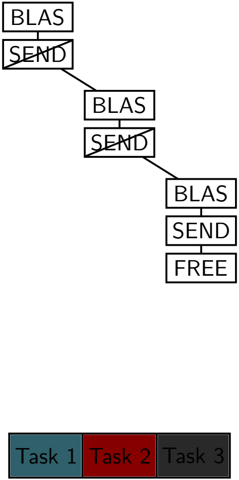


Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full **then**
    Call **Main loop**
end if
Free stuff

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

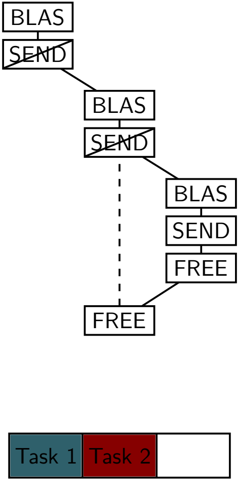


Non-blocking broadcast – implementation

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

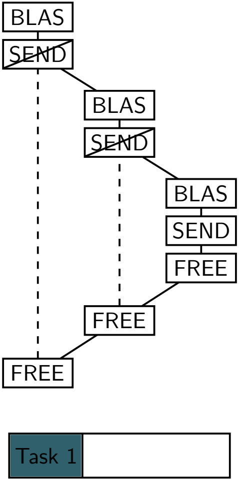


Non-blocking broadcast – implementation

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

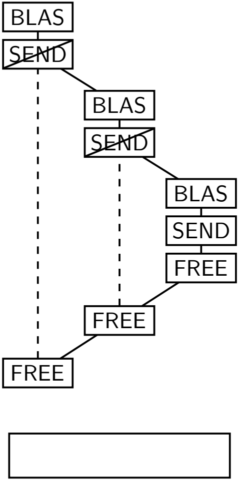


Non-blocking broadcast – implementation

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

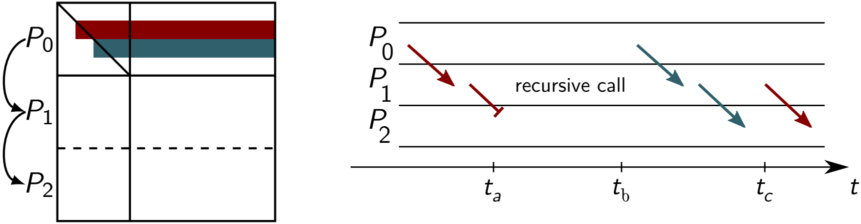
Facto panel:
Do BLAS stuff on panel
Try to send panel to some processes:
if Send buffer is full **then**
  Call **Main loop**
end if
Free stuff



Difficulties:

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

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



P_2 receives the second panel before the first one!

Non-blocking broadcast – experiments

Experiments on a single node of size 64000 with 64 processors:

Strategy	Time (s)	
	1-way threaded BLAS	4-way threaded BLAS
Baseline	596	468
Tree-based	401	167

Better overlapping between communications and computations
$\Rightarrow$ 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.

Memory-aware mapping – more results

Back to experiments with the memory-aware mapping; matrices:

Matrix name	Order (millions)	Entries (millions)	Factors (GB)	S_{seq} (GB)	Description; origin
cage13	0.4	7.5	30.7	17.9	Directed weighted graph; Utrecht U.
as-Skitter	1.7	23.9	17.7	6.2	Internet topology graph; SNAP
HV15R	2.0	283.1	366.4	87.8	CFD, 3D engine fan; FLUOREM
MORANSYS1	2.7	81.3	63.5	19.1	Model Order Reduction; CADFEM
meca_raff6	3.3	130.2	63.5	8.8	Thermo-mechanical coupling; EDF

Memory-aware mapping vs. default strategy (MUMPS 4.10.0):

Matrix	Mapping	S_{max}(MB)	S_{avg}(MB)	Time(s)
cage13	Default	1069	727	285
	MA, $M_0 = 380$MB	305	302	498
as-Skitter	Default	1484	584	578
	MA, $M_0 = 130$MB	135	70	285
HV15R	Default	9063*	8773*	N/A
	MA, $M_0 = 2600$MB	2225	1803	7169
MORANSYS1	Default	1246	834	373
	MA, $M_0 = 380$MB	379	305	651
meca_raff6	Default	1078	796	324
	MA, $M_0 = 320$MB	329	209	367

Conclusion – on-going work

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 $\Longrightarrow$ 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.