qr_mumps

a multithreaded, multifrontal QR solver

The MUMPS team,

MUMPS Users Group Meeting, May 29-30, 2013

The qr_mumps software

qr_mumps

version 1.0

- a multithreaded, multifrontal solver for sparse linear systems based on the QR factorization
- an effort of the MUMPS team
- principal developers
  - Alfredo Buttari
  - + Florent Lopez, Abdou Guermouche, Emmanuel Agullo and George Bosilca (in the GPU branch)
- available under LGPL (CeCILL-C starting from next release) at http://buttari.perso.enseeiht.fr/qr_mumps

qr_mumps

version 1.0

- ~30000 lines of Fortran 2003 code
- OpenMP multithreading
- single/double precision real/complex arithmetic
- multiple ordering tools: SCOTCH, METIS and COLAMD
- singletons detection
- portable C interface (`iso_c_bindings`)
- good documentation ☺

Call sequence

```
program qrm_test
  use qrm_mod
  type(dqrm_spmat_type)          :: qrm_mat

  ...
  call qrm_set(qrm_mat, 'qrm_ordering', qrm_scotch_)
  call qrm_set(qrm_mat, 'qrm_nthreads', 24)
  qrm_mat%icntl(2) = 0

  ! analysis
  call qrm_analyse(qrm_mat)
  ! Q*R=A
  call qrm_factorize(qrm_mat)
  ! Q^T*b
  call qrm_apply(qrm_mat, 't', b)
  ! x=R\b
  call qrm_solve(qrm_mat, 'n', b, x)
end program qrm_test
```

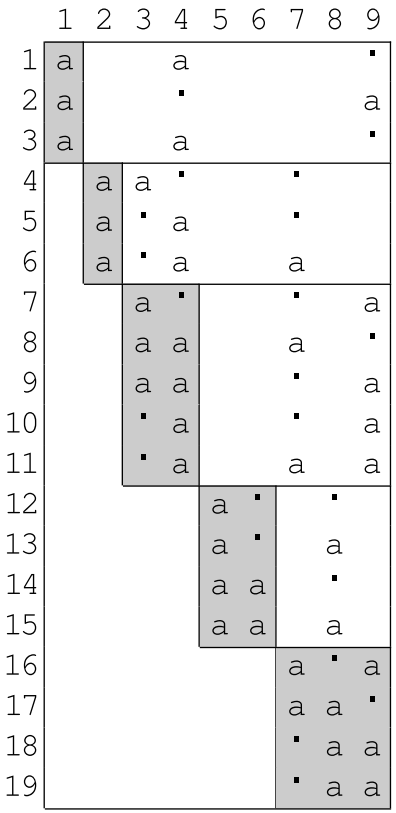
qr_mumps sequence call

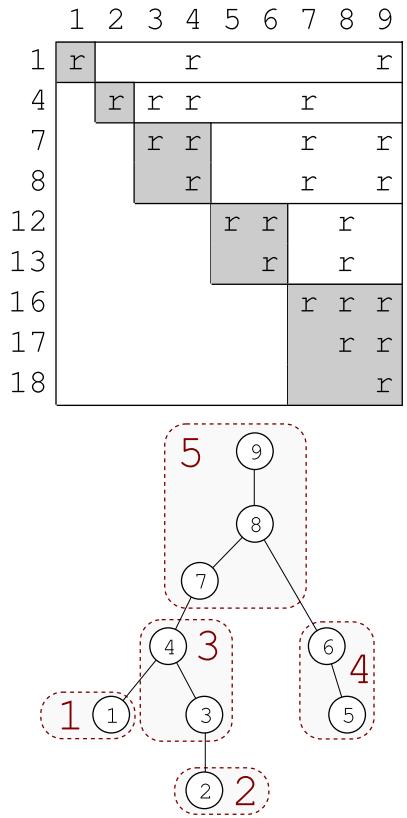
The multifrontal QR method

The Multifrontal QR for newbies

The multifrontal QR factorization is guided by a graph called *elimination tree*:

- at each node of the tree *k* pivots are eliminated
- each node of the tree is associated with a relatively small dense matrix called *frontal matrix* (or, simply, *front*) which contains the *k* columns related to the pivots and all the other coefficients concerned by their elimination





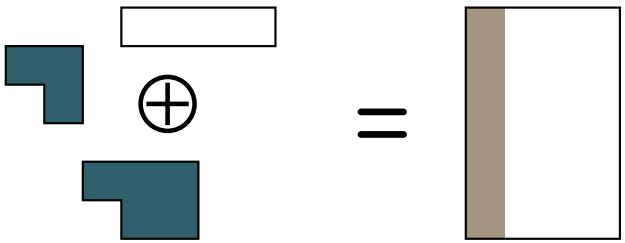
The Multifrontal QR for newbies

The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

The Multifrontal QR for newbies

The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

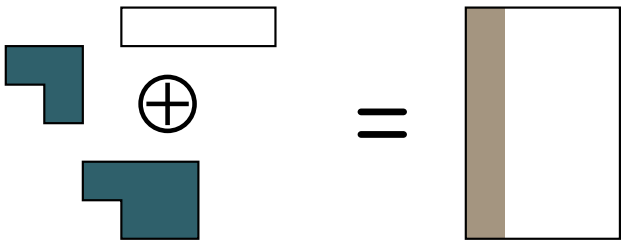
- assembly: a set of coefficient from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix



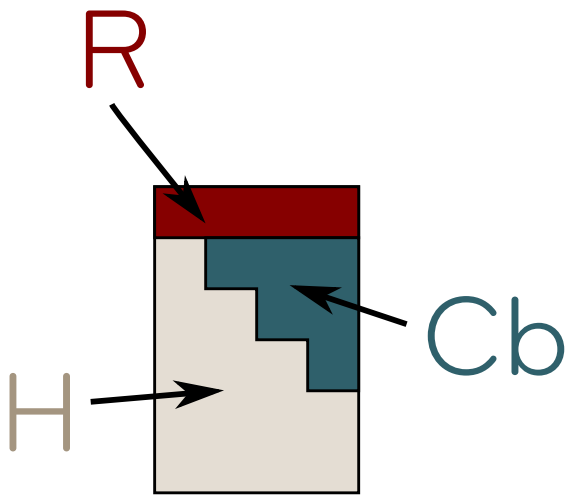
The Multifrontal QR for newbies

The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- assembly: a set of coefficient from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix

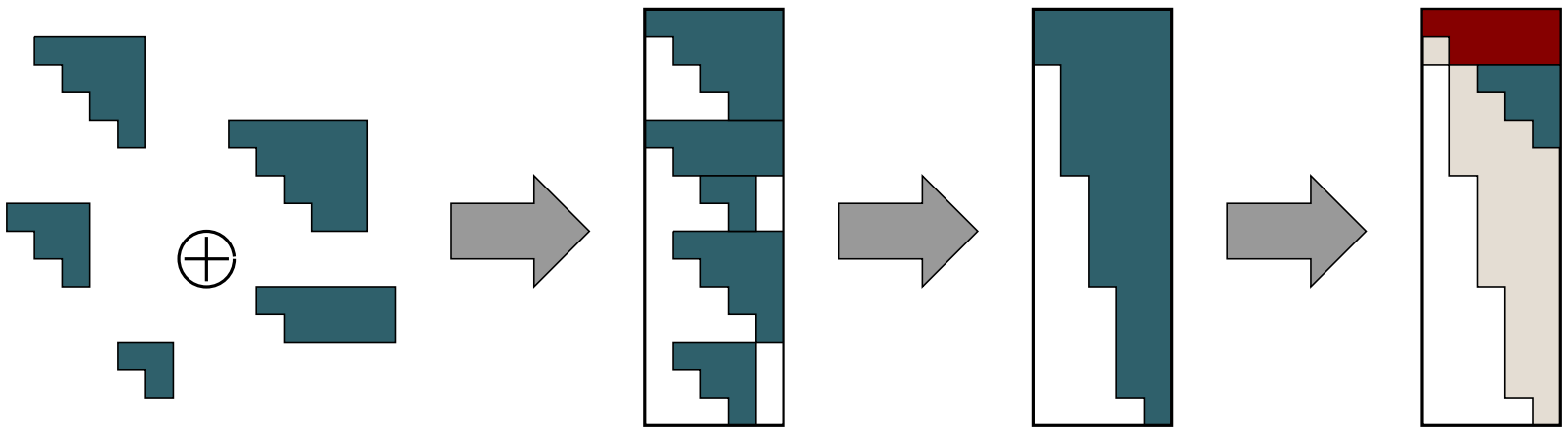


- factorization: the k pivots are eliminated through a complete QR factorization of the frontal matrix. As a result we get:
  - k rows of the global R factor
  - a bunch of Householder vectors
  - a triangular *contribution block* that will be assembled into the father's front



The Multifrontal QR for newbies

Because the contribution blocks are triangular, frontal matrices are (sometimes very) sparse. Rows are thus ordered in increasing order of the first column index.



All the computations and memory related to the bottom left empty part are thus spared

Multifrontal QR, parallelism

The Multifrontal QR: parallelism

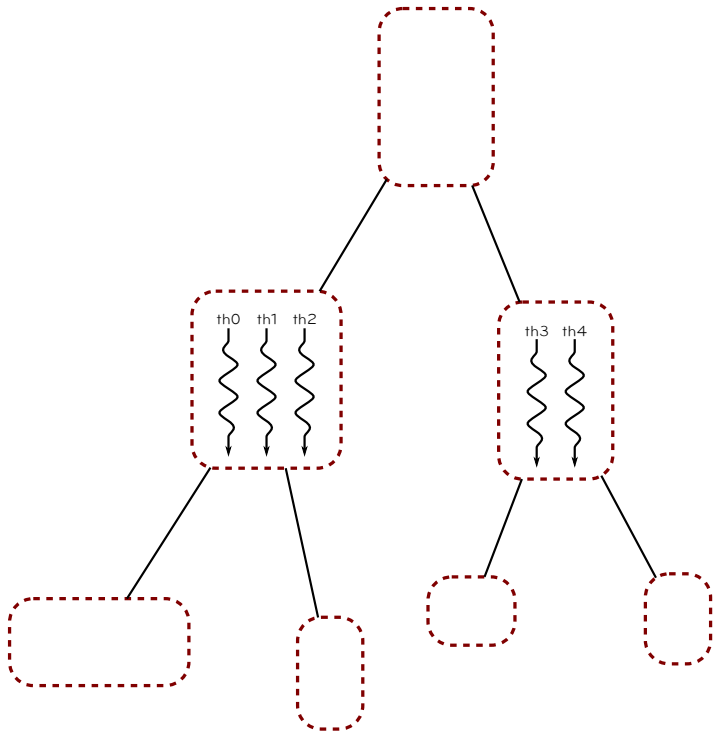
Two sources of parallelism are available in any multifrontal method:

Tree parallelism

- fronts associated with nodes in different branches are independent and can, thus, be factorized in parallel

Front parallelism

- if the size of a front is big enough, multiple processes may be used to factorize it



Parallelism: classical approach

The classical approach (Puglisi, Matstom, Davis)

- Tree parallelism:
  - a front assembly+factorization corresponds to a task
  - computational tasks are added to a task pool
  - threads fetch tasks from the pool repeatedly until all the fronts are done
- Front parallelism:
  - Multithreaded BLAS for the front facto

What's wrong with this approach? A complete separation of the two levels of parallelism which causes

- potentially strong load unbalance
- heavy synchronizations due to the sequential nature of some operations (assembly)
- sub-optimal exploitation of the concurrency in the multifrontal method

Parallelism: a new approach

fine-grained, data-flow parallel approach

- fine granularity: tasks are not defined as operations on fronts but as operations on portions of fronts defined by a 1-D partitioning
- data flow parallelism: tasks are scheduled dynamically based on the dependencies between them

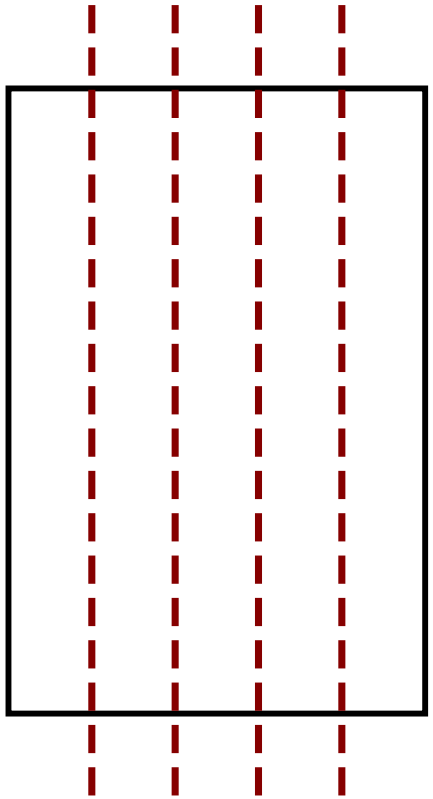
Both node and tree parallelism are handled the same way at any level of the tree.

Fine grained, asynchronous, parallel QR

Parallelism: a new approach

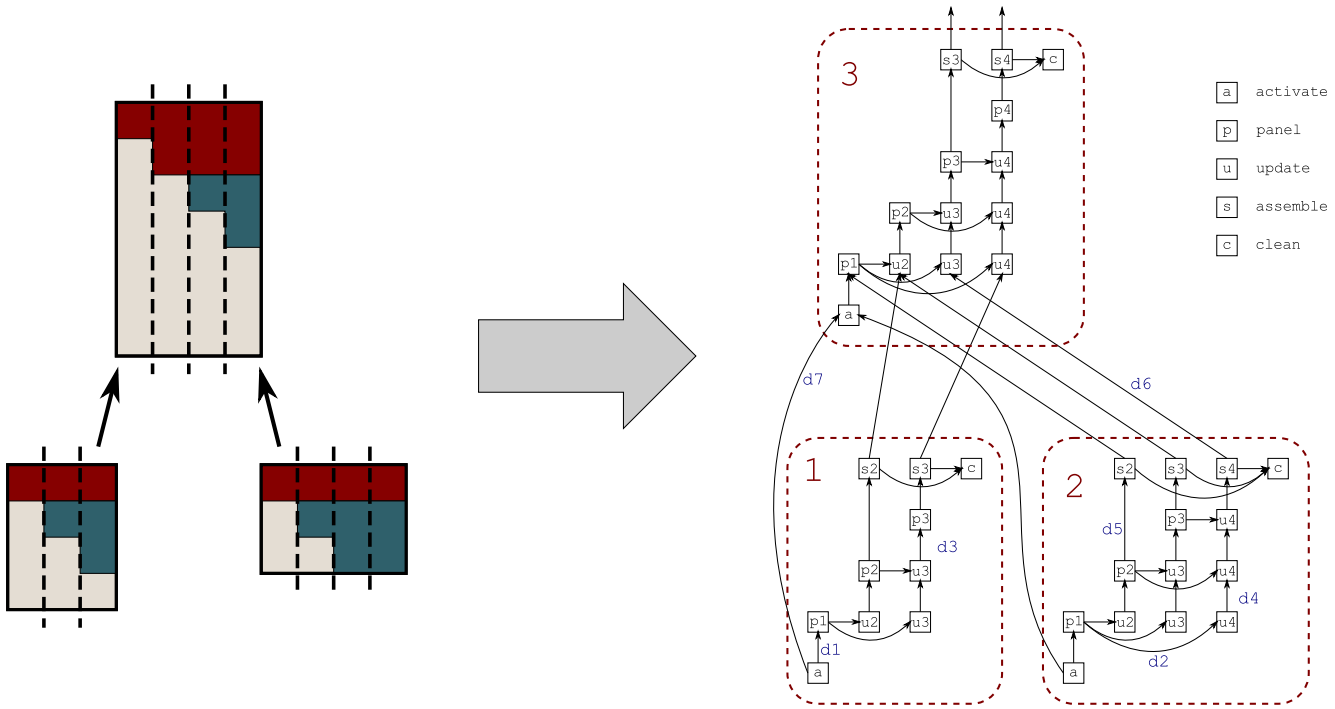
Fine-granularity is achieved through a 1-D block partitioning of fronts and the definition of five elementary operations:

1. `activate(front)`: the activation of a front corresponds to a full determination of its (staircase) structure and allocation of the needed memory areas
2. `panel(bcol)`: QR factorization (Level2 BLAS) of a column
3. `update(bcol)`: update of a column in the trailing submatrix wrt to a panel
4. `assemble(bcol)`: assembly of a column of the contribution block into the father
5. `clean(front)`: cleanup the front in order to release all the memory areas that are no more needed



Parallelism: a new approach

If a task is defined as the execution of one elementary operation on a block-column or a front, then the entire multifrontal factorization can be represented as a Directed Acyclic Graph (DAG)



where the nodes represent tasks and edges the dependencies among them

Parallelism: a new approach

- d1: no other elementary operation can be executed on a front or on one of its block-columns until the front is not activated;
- d2: a block column can be updated with respect to a panel only if the corresponding panel factorization is completed;
- d3: the **panel** operation can be executed on block-column i only if it is up-to-date with respect to panel $i-1$;
- d4: a block-column can be updated with respect to a panel i in its front only if it is up-to-date with respect to the previous panel $i-1$ in the same front;
- d5: a block-column can be assembled into the parent (if it exists) when it is up-to-date with respect to the last panel factorization to be performed on the front it belongs to (in this case it is assumed that block-column i is up-to-date with respect to panel i when the corresponding **panel** operation is executed);
- d6: no other elementary operation can be executed on a block-column until all the corresponding portions of the contribution blocks from the child nodes have been assembled into it, in which case the block-column is said to be *assembled*;
- d7: since the structure of a frontal matrix depends on the structure of its children, a front matrix can be activated only if all of its children are already active;

Parallelism: a new approach

Tree

\+

Node

DAG

Advantages of the DAG approach:

- no need for performance modeling (i.e., "how many BLAS threads should I put on this front?")
- more concurrency because it is possible to work on a node before its children are completely factorized (because its block-columns are assemble separately)
- no problems with technology (it is often difficult to use multithreading both outside and inside BLAS routines)

Parallelism: a new approach

Tasks execution is dynamically triggered by a basic scheduler based on task queues:

Main loop

```
mainloop: do
   if(n_ready_tasks < ntmin) then
      ! if the number of threads falls
      ! below a certain value, fill-up
      ! the queues
      call fill_queues()
   end if

  tsk = pick_task()
  select case(tsk)
  case(panel)
     call execute_panel()
  case(update)
     call execute_update()
  case(assemble)
     call execute_assemble()
  case(activate)
     call execute_activate()
  case(clean)
     call execute_clean()
  case(finish)
     exit mainloop
end do
```

fill_queues()

```
found = .false.

forall (front in active fronts)
   ! for each active front try to schedule
   ! ready tasks
   found = found .or. push_panels(front)
   found = found .or. push_updates(front)
   found = found .or. push_assembles(front)
   found = found .or. push_clean(front)
end forall

if (found) then
   ! if tasks were pushed in the previous
   ! loop return
   return
else
   ! otherwise schedule the activation of
   ! the next ready front
   call push_activate(next ready front)
end if

if (factorization over) call push_finish()
```

Scheduling

A few comments on scheduling:

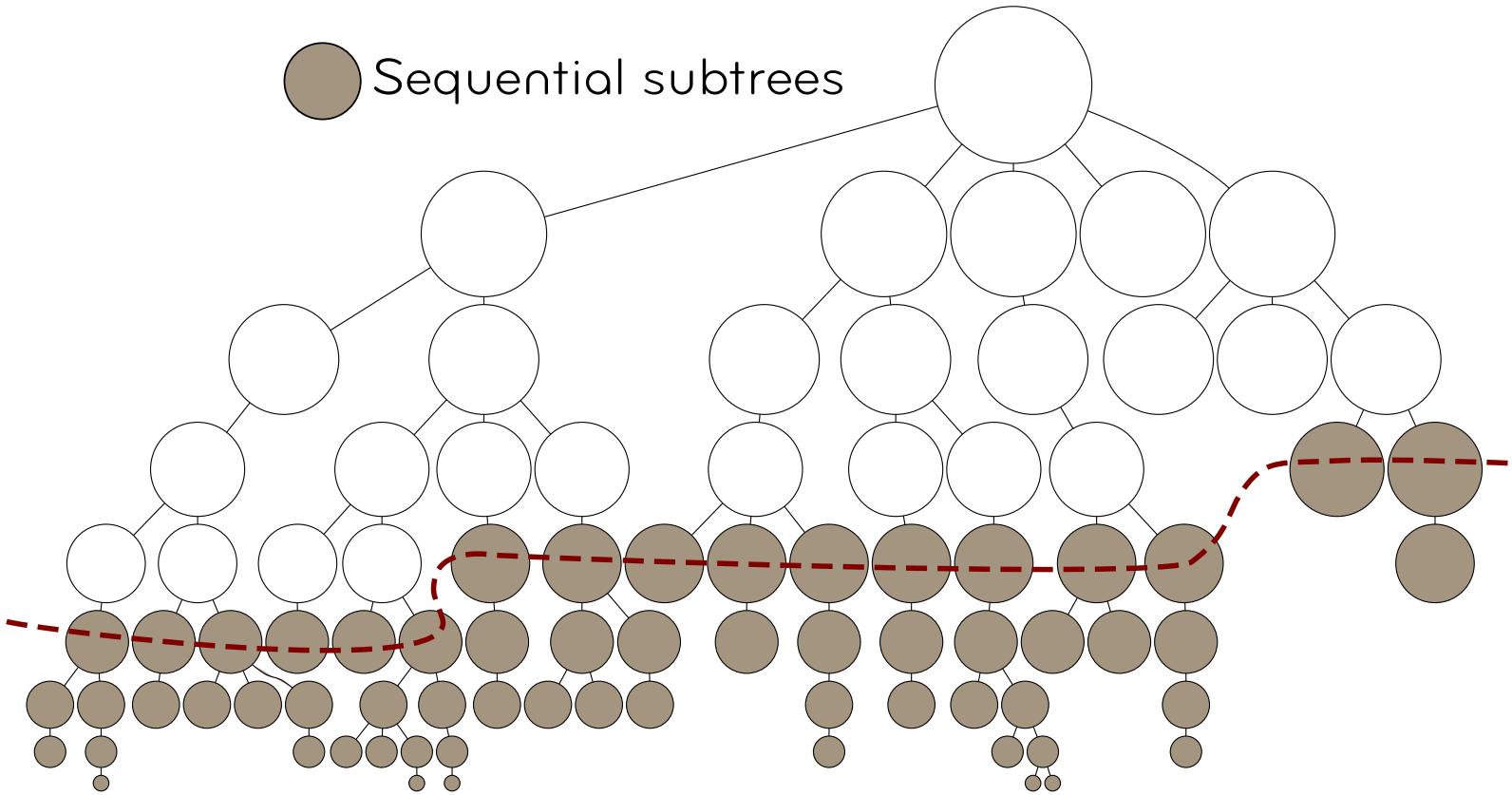
- fronts are activated according to a pre-computed order (for example on that minimizes the memory consumption)
- the scheduling follows this rule of thumb:

  "as long as there is work to do on active fronts, no other front should be activated"

  This allows to follow the pre-computed ordering more closely
- tasks which lie along the critical path are given higher priority in order to reduce the time to completion
- other "smart" schedulings are possible, e.g., in order to cope with NUMA memory layouts
- the size of the search-space (for tasks) depends on the number of active fronts

Scheduling: logical pruning

Because the number of nodes in the tree is much larger than the number of cores used, entire subtrees are scheduled at once in order to reduce the scheduling complexity:



Scheduling: tree reordering

Assuming nc_i is the number of children of node i, the maximum number of active nodes when traversing the subtree rooted at i is

$$P_i = max(\overbrace{max_{j=1,\ldots,nc_i}(j-1+P_j)}^{1}, \overbrace{nc_i+1}^{2})$$

where

Minimizing all the P_i is achieved by sorting the nodes in ascending order of nc_i

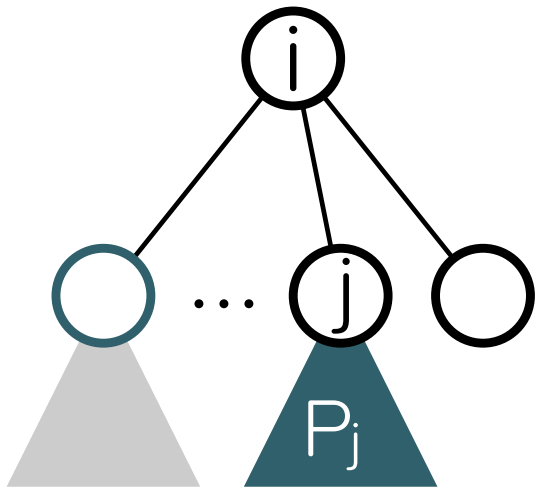
Scheduling: tree reordering

Assuming nc_i is the number of children of node i, the maximum number of active nodes when traversing the subtree rooted at i is

$$P_i = max(\overbrace{max_{j=1,\ldots,nc_i}(j-1+P_j)}^{1}, \overbrace{nc_i+1}^{2})$$

where

1. the max number of active nodes while its children are being treated



Minimizing all the P_i is achieved by sorting the nodes in ascending order of nc_i

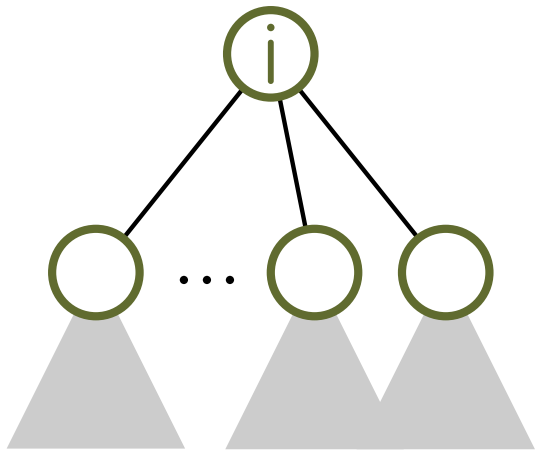
Scheduling: tree reordering

Assuming nc_i is the number of children of node i, the maximum number of active nodes when traversing the subtree rooted at i is

$$P_i = max(\overbrace{max_{j=1,\ldots,nc_i}(j - 1 + P_j)}^{1}, \overbrace{nc_i + 1}^{2})$$

where

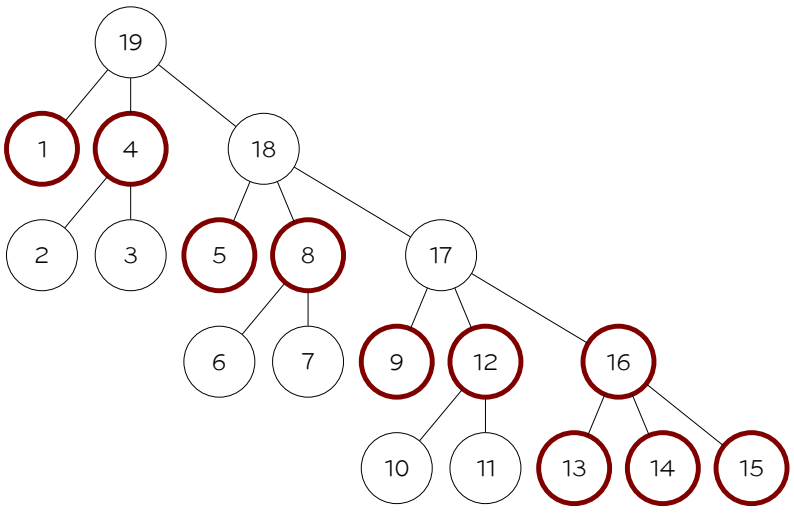
1. the max number of active nodes while its children are being treated
2. is the number of active nodes when i is being treated

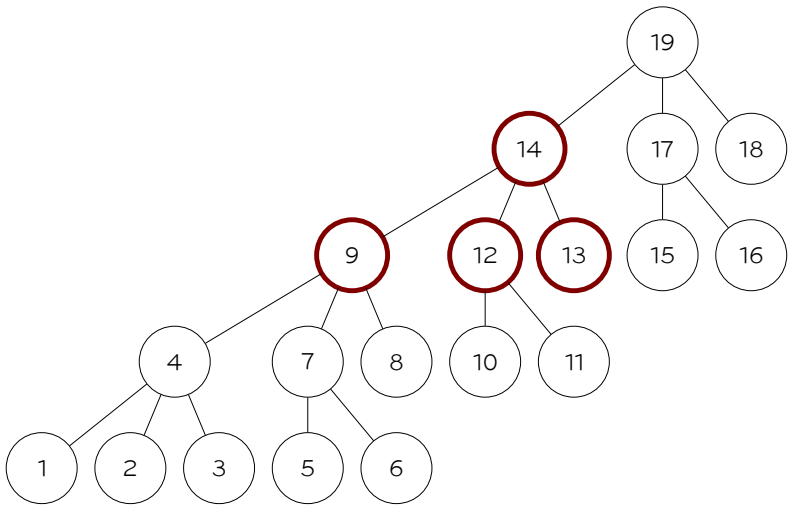


Minimizing all the P_i is achieved by sorting the nodes in ascending order of nc_i

Scheduling: tree reordering

Example:



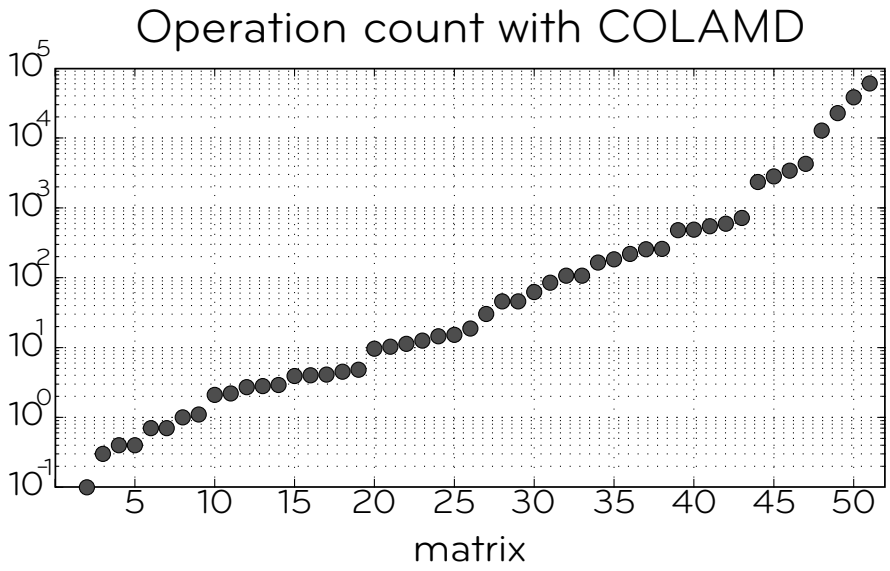


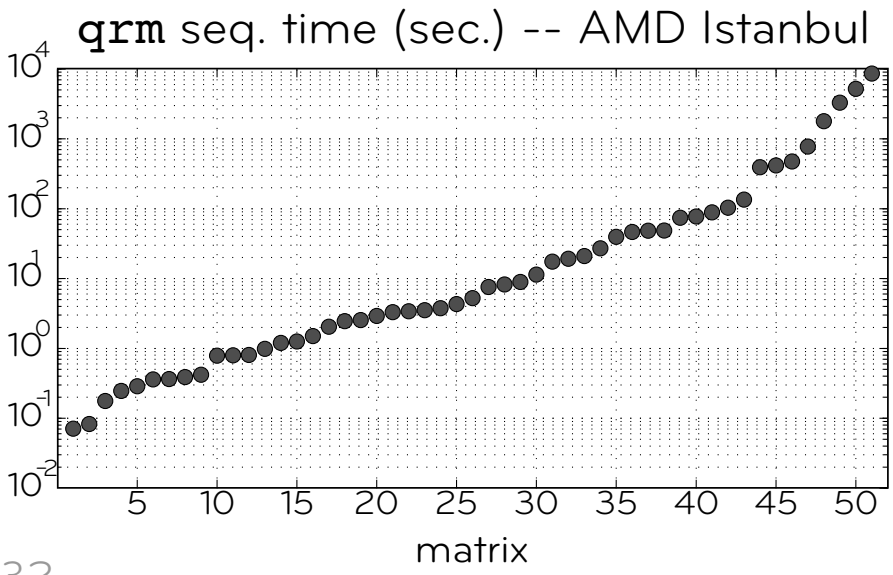
Using the postorder on the left $P_{19} = 10$ whereas, after reordering, $P_{19} = 4$.

Experimental results

Experimental results

51 matrices from UF colection and an AMD Istanbul computer

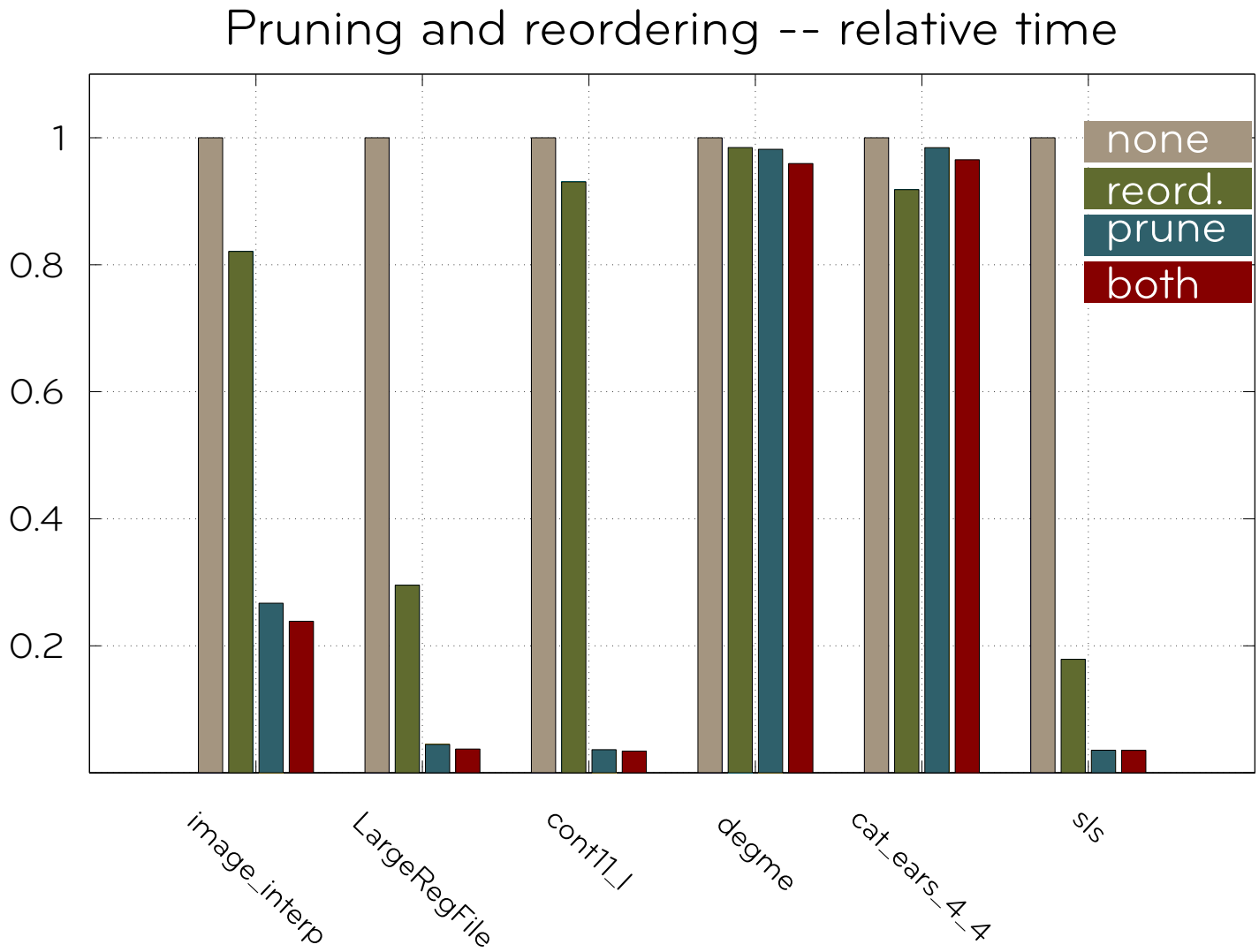




System	AMD Istanbul
total # of cores	24 (6×4)
freq.	2.4 GHz
mem. type	NUMA
compilers	Intel 11.1
BLAS/LAPACK	Intel MKL 10.2

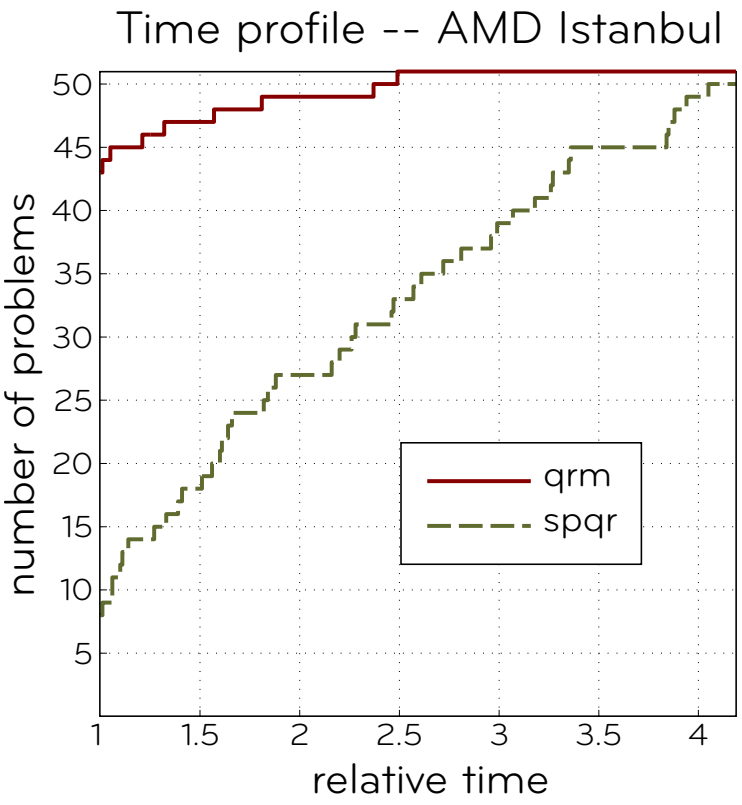
Experimental results

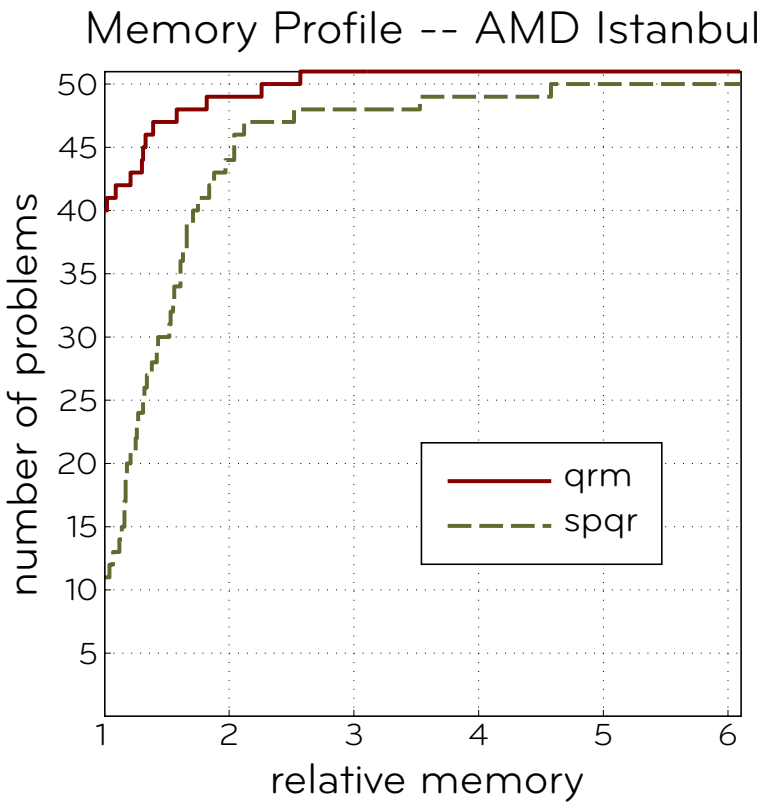
The effect of reordering and pruning



Experimental results

Comparison with SPQR:

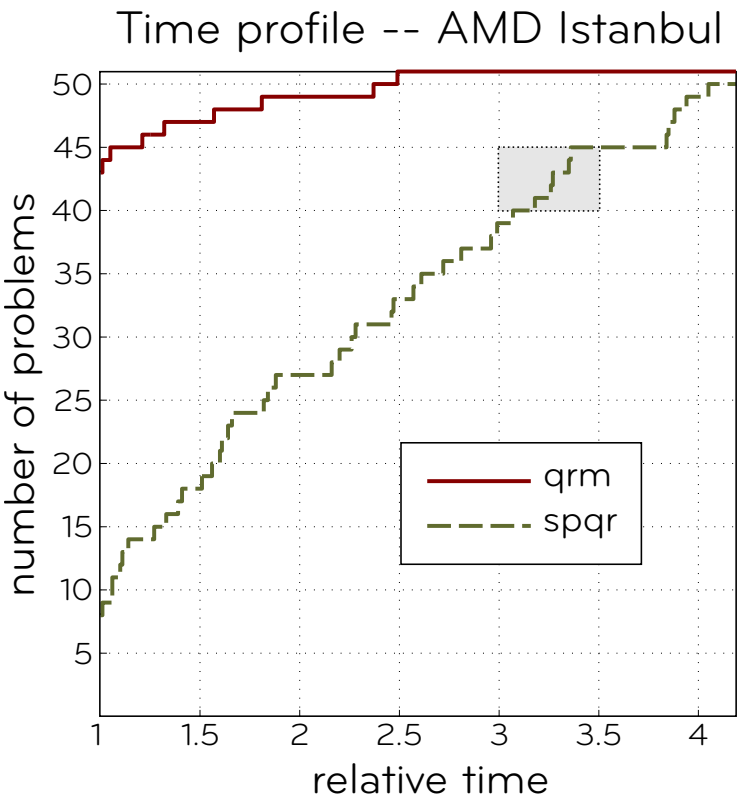


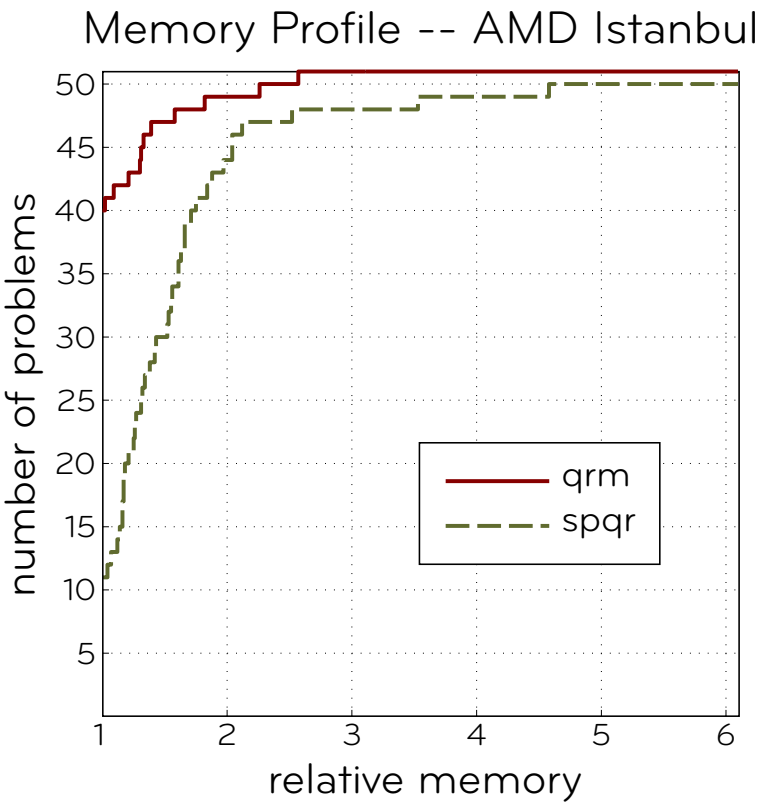


a data point (x, y) means that the code is no worse than x times the best of the two codes for y problems

Experimental results

Comparison with SPQR:





a data point (x, y) means that the code is no worse than x times the best of the two codes for y problems

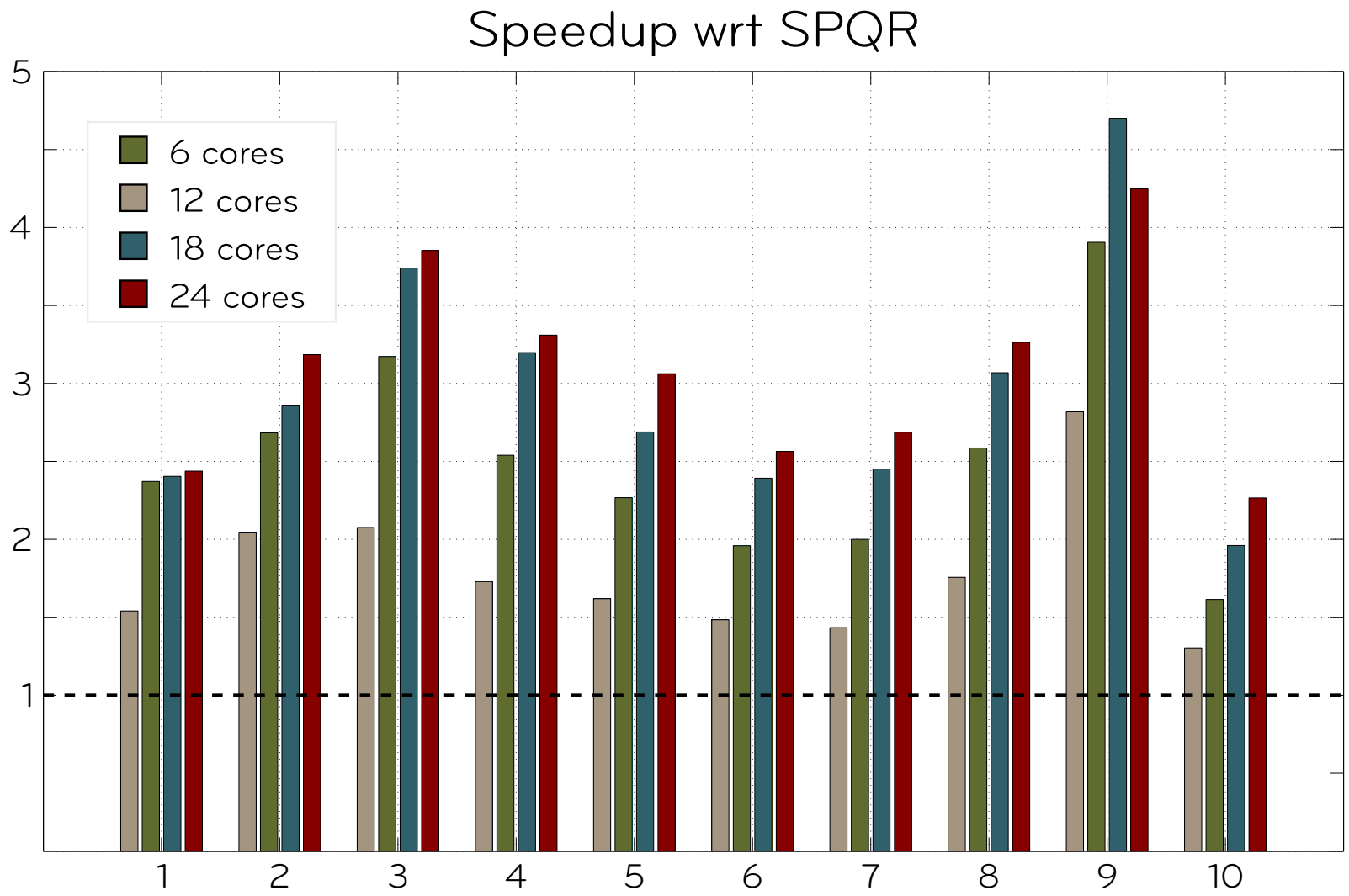
Experimental results

Comparison with SPQR – details:

#	Mat. name	m	n	nz	op. count (Gflops)
1	cont11_l	1468599	1961395	5382999	184.5
2	EternityII_E	11077	262144	1572792	544.0
3	degme	185501	659415	8127528	591.9
4	cat_ears_4_4	19020	44448	132888	716.1
5	Hirlam	1385270	452200	2713200	2339.9
6	e18	24617	38602	156466	3399.5
7	flower_7_4	27693	67,593	202218	4261.2
8	Rucci1	1977885	109900	7791168	12768.5
9	sls	1748122	62729	6804304	22716.2
10	TF17	38132	48630	586218	38203.1

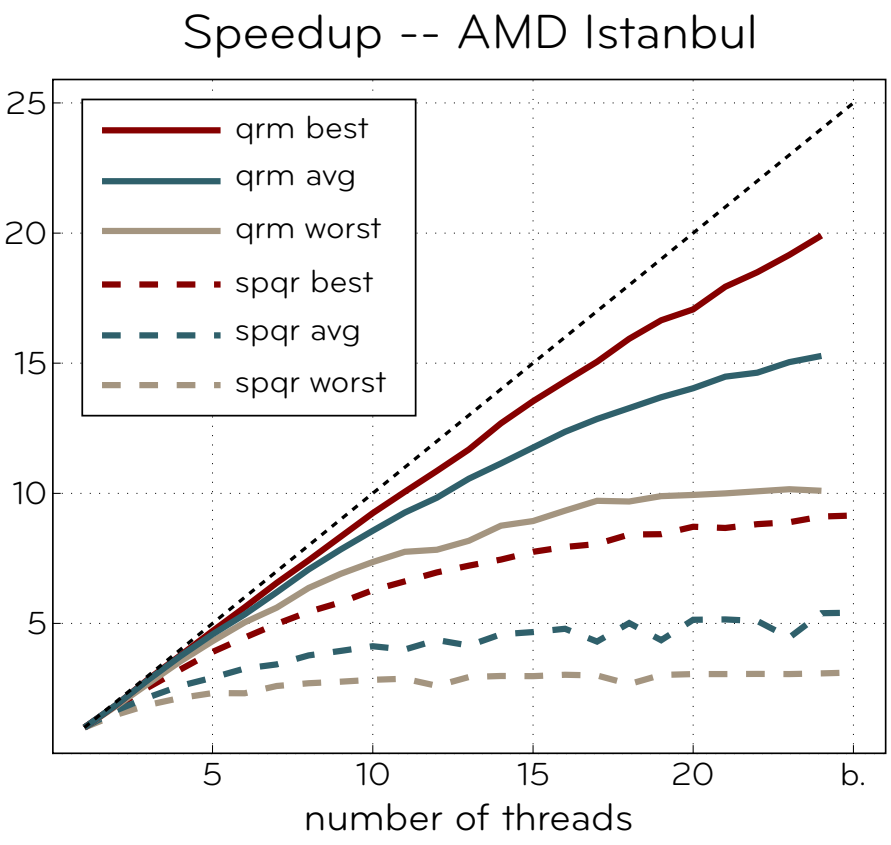
Experimental results

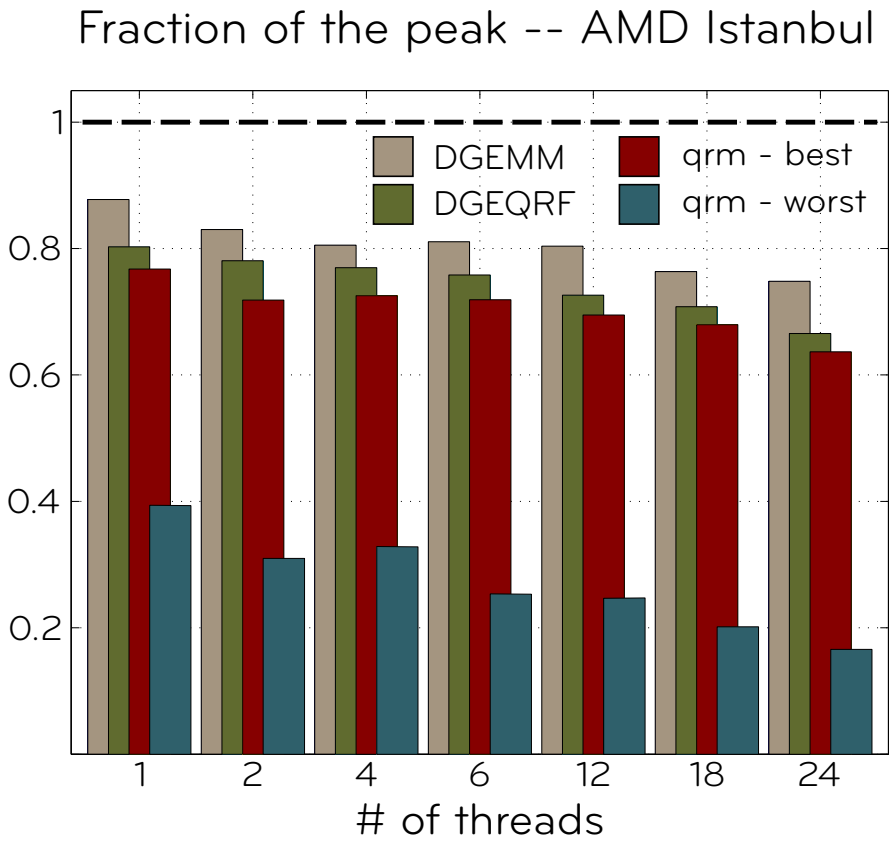
Comparison with SPQR – details:



Experimental results

Comparison with SPQR – scalability:





Reference

- A. Buttari.
  Fine-grained multithreading for the multifrontal QR factorization of sparse matrices.
  To appear on the SIAM Journal on Scientific Computing.

- `qr_mumps 1.0` User's Guide
  `http://buttari.perso.enseeiht.fr/qr_mumps`

Ongoing and future work

- replace the basic scheduler with a full-featured, modern runtime system (Florent with Abdou and Emmanuel, see next talk)
- port to accelerated (GPUs, MICs etc.) platforms (same people as above)
- 2D fronts partitioning (for more parallelism)
- distributed memory systems ?
- memory-constrained schedulings
- rank-deficient matrices and/or rank-revealing QR factorization

Thank you all!

Questions?