# Sparse QR Factorizations

in multicore sauce

Alfredo Buttari, CNRS-IRIT Toulouse

MUMPS Users Group Meeting, April 2010

## Multifrontal QR, introduction

MUMPS

## Multifrontal QR

The Multifrontal QR method builds upon the equivalence between the $R$ factor and the Cholesky factor of $A^TA$

**From $A^TA = LL^T$ to $A = QR$**

Under the assumption that $A$ is a Strong Hall matrix, $L$ and $R$ have exactly the same structure.

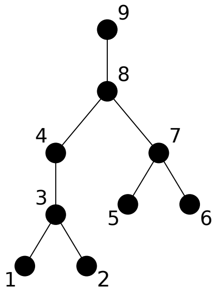
A **Multifrontal** method can be used relying on the elimination/assembly tree generated for the Cholesky factorization of $A^TA$

## Multifrontal QR

1. Analysis: symbolic computations to reduce fill-in, compute elimination tree, symbolic factorization, estimates etc.
2. Factorization: compute the $Q$ and $R$ factors
3. Solve: use $Q$ and $R$ to compute the solution of a problem (e.g. $min_x \|Ax - b\|_2$)

## Multifrontal QR

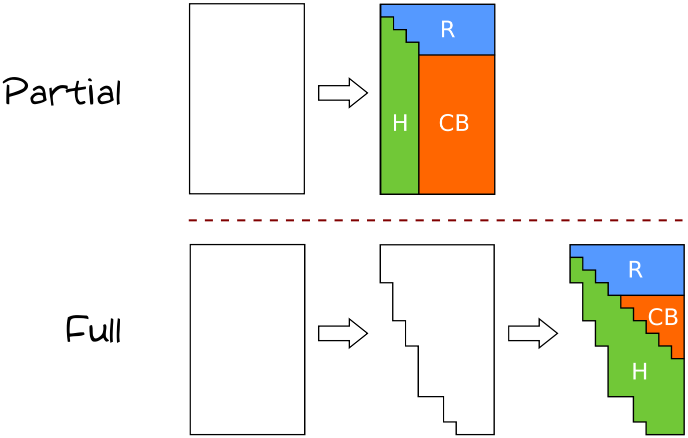
2. Factorization: compute the $Q$ and $R$ factors



- the tree is processed bottom-up
- a dense frontal matrix is associated to each node
- at each node:
  1. Assembly: the contribution blocks from the children nodes are assembled into the frontal matrix
  2. Factorization: the frontal matrix is factorized (fully or partially)

# The Multifrontal QR factorization

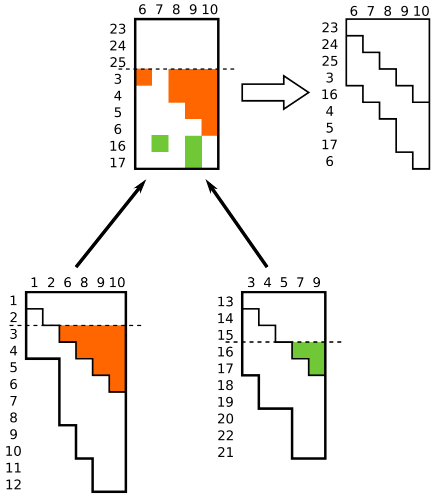
Different approaches can be used for front factorization:



Option 2 (Strategy 3 in Puglisi's thesis) is the winner.

## The Multifrontal QR factorization

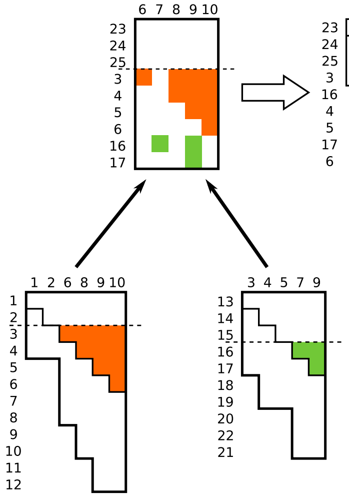
Things get complicated when we look at all the fronts together.



# The Multifrontal QR factorization

Things get complicated when we look at all the fronts together.

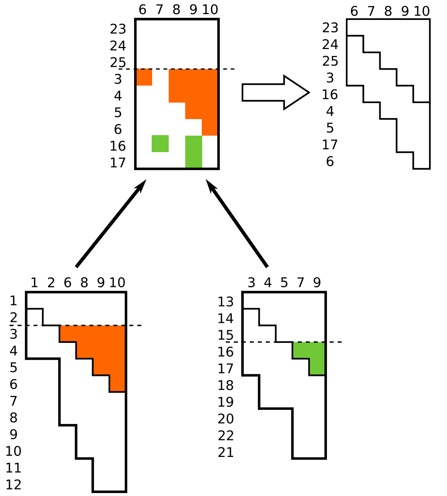
1. contribution blocks are simply appended at the bottom of the father front



# The Multifrontal QR factorization

Things get complicated when we look at all the fronts together.

1. contribution blocks are simply appended at the bottom of the father front
2. a row permutation must be computed to restore a staircase structure



## Multifrontal QR, parallelism

## Parallelism

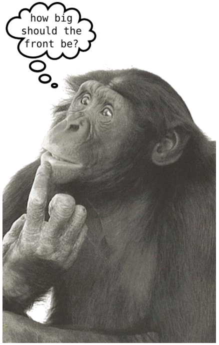
As for the Cholesky, $LU$, $LDL^T$ multifrontal method, two levels of parallelism can be exploited:

- Tree Parallelism: fronts associated to nodes in different branches are independent and can, thus, be factorized in parallel
- Front Parallelism: if the size of a front is big enough, the front can be factorized in parallel

## Parallelism

As for the Cholesky, $LU$, $LDL^T$ multifrontal method, two levels of parallelism can be exploited:

- Tree Parallelism: fronts associated to nodes in different branches are independent and can, thus, be factorized in parallel
- Front Parallelism: if the size of a front is big enough, the front can be factorized in parallel



## Parallelism: classical approach

### The classical approach (Puglisi, Matstom, Davis)

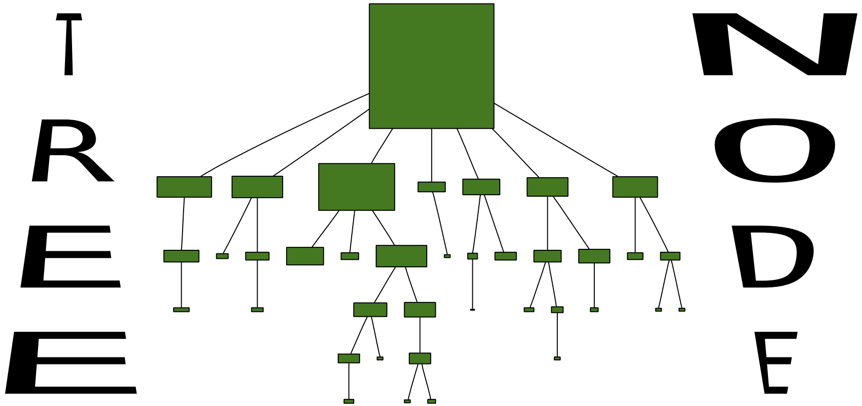
- Tree parallelism:
  - a front assembly+factorization corresponds to a task
  - computational tasks are added to a task pool
  - threads fetch tasks from the pool repeatedly until all the fronts are done
- Node parallelism:
  - Multithreaded BLAS for the front facto

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: classical approach

Node parallelism grows towards the root



Tree parallelism grows towards the leaves

## Parallelism: a new approach

**fine-grained, data-flow parallel approach**

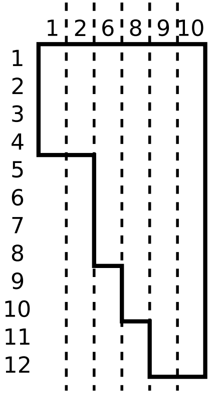
- fine granularity: tasks are not defined as operations on fronts but as operations on portions of fronts defined by a 1-D partitioning
- data flow parallelism: tasks are scheduled dynamically based on the dependencies between them

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

## 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



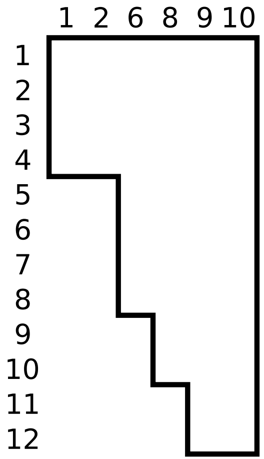
## Parallelism: a new approach

How do we handle all this complexity?

**Data-flow programming model [Silc et al. 97]**

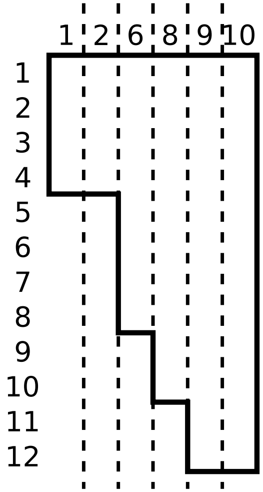
*An instruction is said to be executable when all the input operands that are necessary for its execution are available to it. The instruction for which this condition is satisfied is said to be fired. The effect of Firing an instruction is the consumption of its input values and generation of output values.*

## Parallelism: a new approach



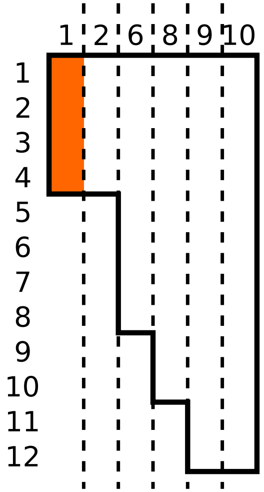
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns



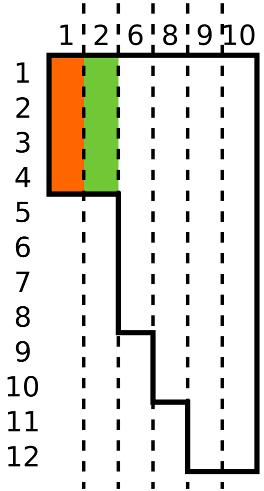
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual



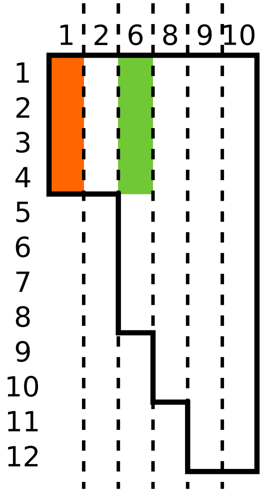
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



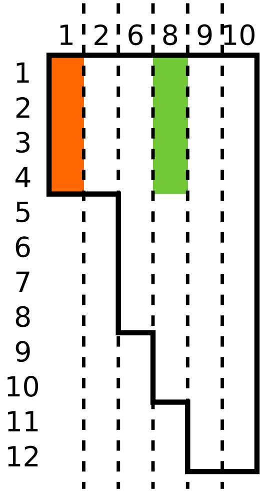
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



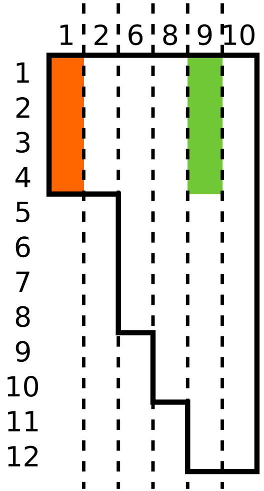
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



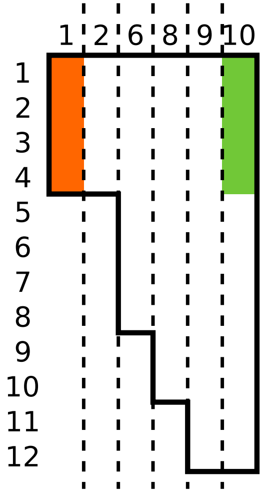
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



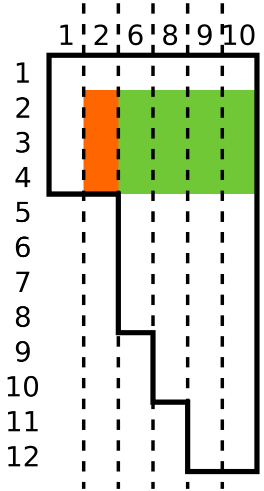
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



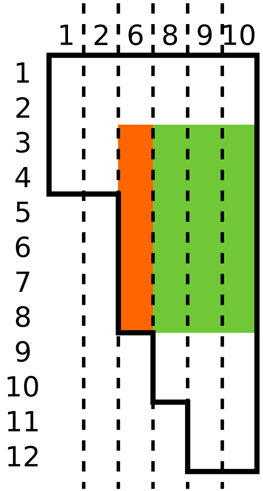
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



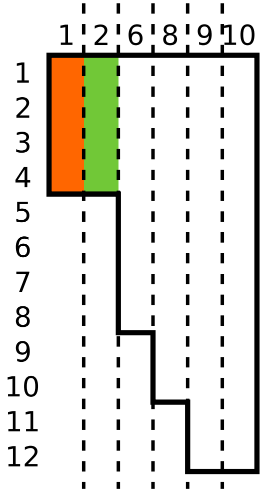
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



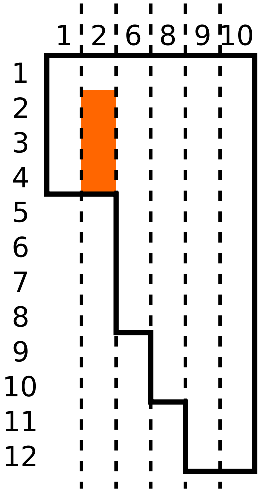
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately



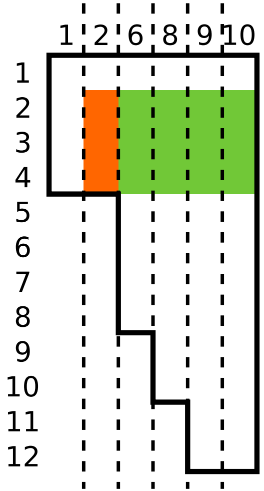
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately
- Firing rule #1: a panel can be factorized as soon as it is updated wrt the previous step (lookahead). Early panel factorizations results in more updates in a "ready" state and, thus, more parallelism



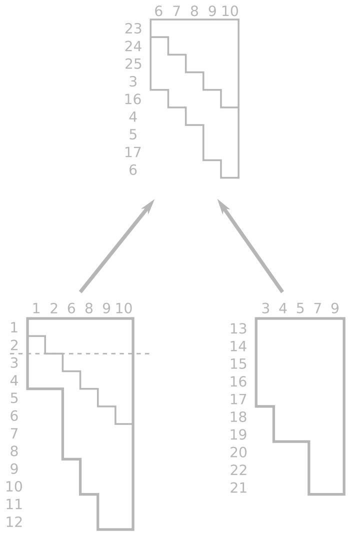
## Parallelism: a new approach

- a frontal matrix is 1-D partitioned into block-columns
- panels are factorized as usual
- updates can be applied to each column separately
- Firing rule #1: a panel can be factorized as soon as it is updated wrt the previous step (lookahead). Early panel factorizations results in more updates in a "ready" state and, thus, more parallelism
- Firing rule #2: a column can be updated wrt a panel if it is up to date wrt all previous panels



## Parallelism: a new approach

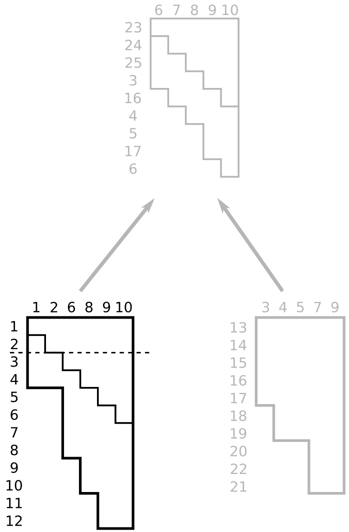
A look at the whole tree:



## Parallelism: a new approach

A look at the whole tree:

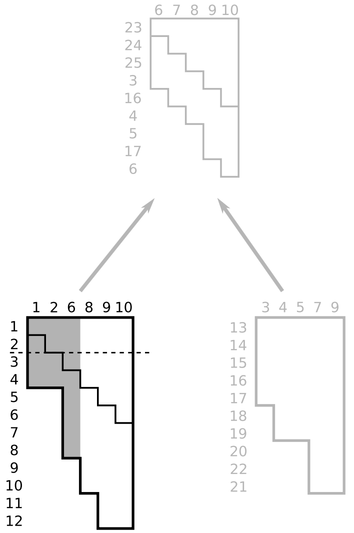
- fronts must be activated: the structure is computed and memory is allocated



## Parallelism: a new approach

A look at the whole tree:

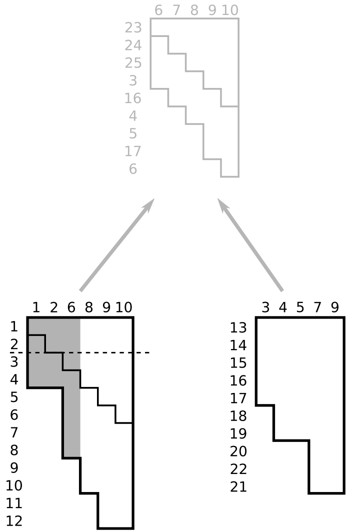
- fronts must be activated: the structure is computed and memory is allocated



## Parallelism: a new approach

A look at the whole tree:

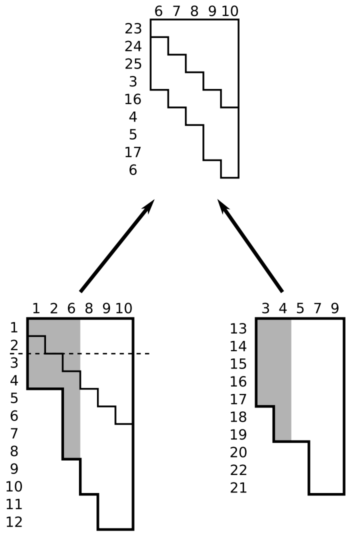
- fronts must be activated: the structure is computed and memory is allocated



## Parallelism: a new approach

A look at the whole tree:

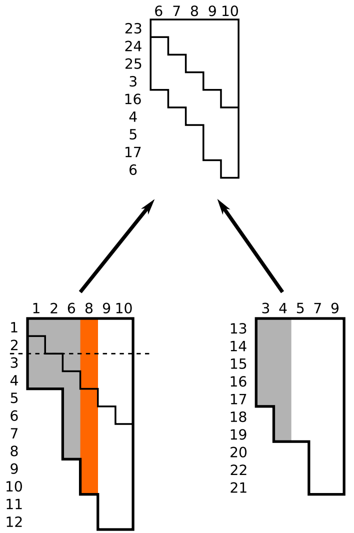
- fronts must be activated: the structure is computed and memory is allocated
- Firing rule #3: a node can be activated only if all of its children are already active



## Parallelism: a new approach

A look at the whole tree:

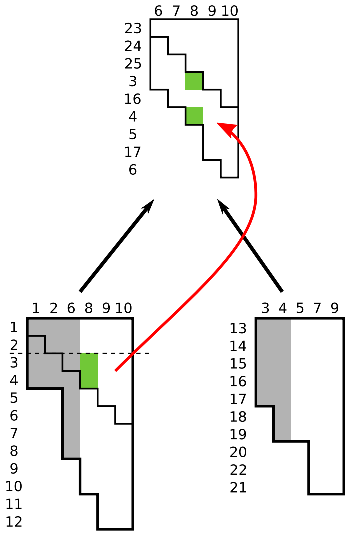
- fronts must be activated: the structure is computed and memory is allocated
- Firing rule #3: a node can be activated only if all of its children are already active



# Parallelism: a new approach

A look at the whole tree:

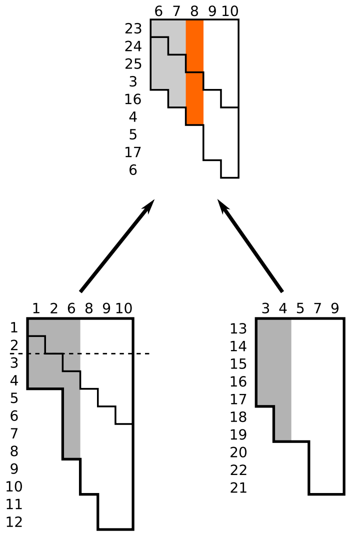
- fronts must be activated: the structure is computed and memory is allocated
- Firing rule #3: a node can be activated only if all of its children are already active
- Firing rule #4: a column can be assembled into the father, if it is up-to-date wrt all the preceding panels and the father is active



## Parallelism: a new approach

A look at the whole tree:

- fronts must be activated: the structure is computed and memory is allocated
- Firing rule #3: a node can be activated only if all of its children are already active
- Firing rule #4: a column can be assembled into the father, if it is up-to-date wrt all the preceding panels and the father is active
- Firing rule #5: a column can be factorized if it is fully assembled regardless of the status of the rest of the front



## Parallelism: the DAG-tree

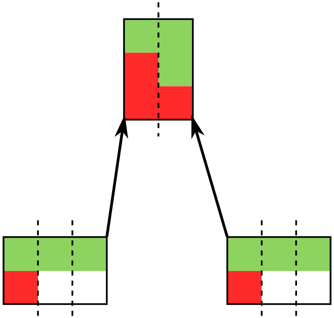
**Data-flow programming model [Silc et al. 97]**

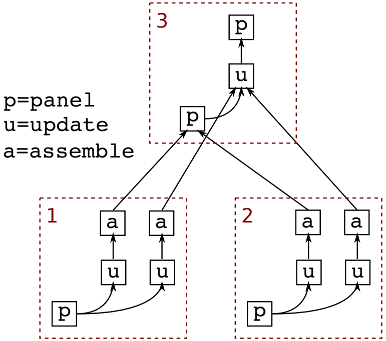
*As a result, a dataflow program can be represented as a directed graph consisting of named nodes, which represent instructions, and arcs, which represent data dependencies among instructions.*

## Parallelism: the DAG-tree

**Data-flow programming model [Silc et al. 97]**

*As a result, a dataflow program can be represented as a directed graph consisting of named nodes, which represent instructions, and arcs, which represent data dependencies among instructions.*





## Parallelism: scheduling

exec_loop
```
do

   call get_task()

   select case(task_type)
   case (0)
      exit
   case (1)
      call do_activate(...)
   case (2)
      call do_panel(...)
   case (3)
      call do_update(...)
   case (4)
      call do_assemble(...)
   case (5)
      call do_clean(...)
   end select

end do
```

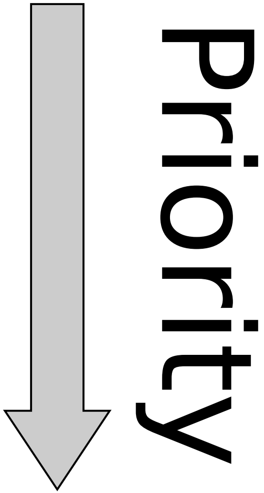
**Data-flow programming model [Silc et al. 97]**

*Due to the above rule the model is asynchronous. It is also self-scheduling since instruction sequencing is constrained only by data dependencies.*

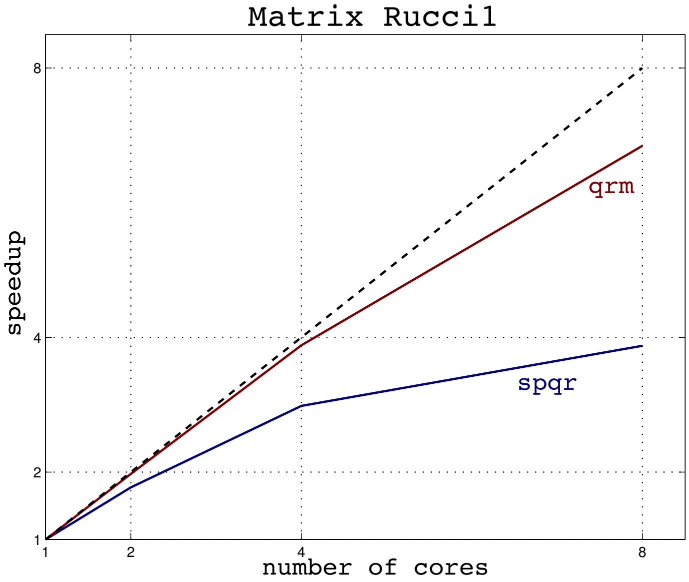
## Parallelism: scheduling

get_task

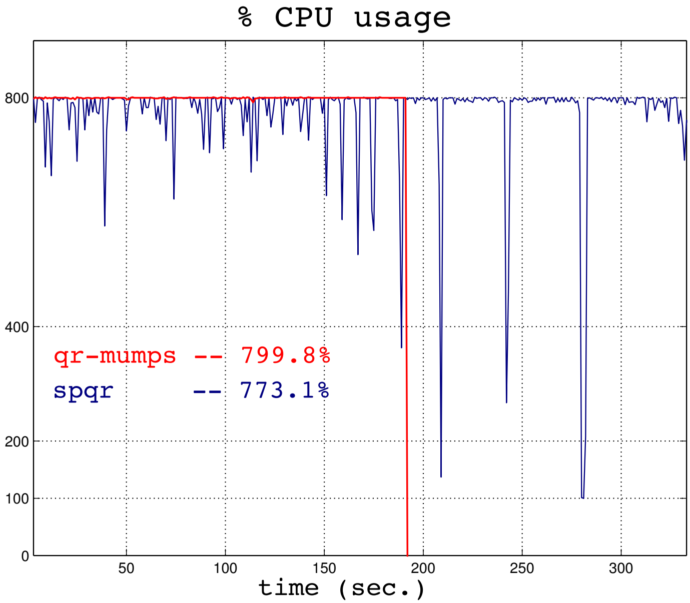
```
do

   do f=1, num_fronts

      if (f is active) then
         call get_panel()     ! set task_type=2
         call get_update()    ! set task_type=3
         call get_assemble()  ! set task_type=4
      else if (f is activable) then
         task_type=1
      else if (f is done) then
         task_type=5
      end if

   end do

   if (factorization done) task_type=0
   if (found task) exit

end do
```



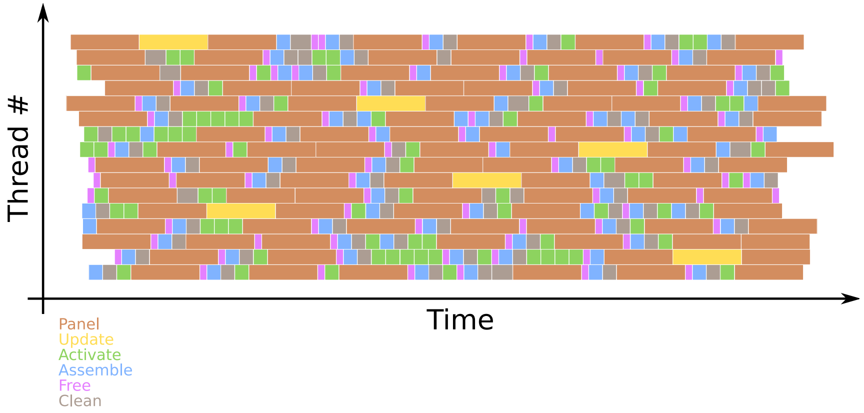
## Parallelism: results



## Parallelism: results



## Parallelism: results



## QR-MUMPS

## QR-MUMPS

**QR-MUMPS: keywords**

QR, least-squares, multifrontal, hybrid parallelism, asynchronous, rank-revealing, Fortran, free, open-source

The idea to develop QR-MUMPS stems from:

- the experience accumulated in MUMPS (by Patrick, Jean-Yves and Abdou)
- the enthusiasm (and time) of young researchers (Alfredo, Bora)
- a solid base in the work done by Chiara
- the RTRA project

## QR-MUMPS features

Done!

COLAMD Ordering, Symbolic Factorization, OpenMP factorization, Singletons Detection, Amalgamation, Fortran 95/2003 software infrastructure, stackless memory management, multiple precisions

## QR-MUMPS features

TODO

Solution, Rank Revealing, MPI tree parallelism, MPI front parallelism, reorder tree, front-to-processor mapping, memory consumption minimization, more ordering methods, splitting, in-place assembly flops/memory estimates, matlab interface,out-of-core, numerical preprocessing, C interface, blocking optimality, low-rank approximations, 2-D OpenMP parallelism, memory affinity, scheduling policies, parallel analysis, partial QR, incomplete QR

## The end

```
do
   write(*,'(" Questions?")')

   if (question) then
      if(have_answer) then
         call give_answer()
      else
         call pretend_the_question_is_ill_posed()
      end if
   else
      write(*,'(" Thank you!")')
      exit
   end if
end do
```