QR_MUMPS: PROGRESS THROUGH THE SOLHAR PROJECT

E. Agullo, A. Buttari, F. Flopez, A. Guermouche, A. Legrand, I. Masliah and L. Stanisic

Plenary Solharis meeting, February 5, 2020, Bordeaux

Linear Systems and Direct Methods

Sparse linear systems

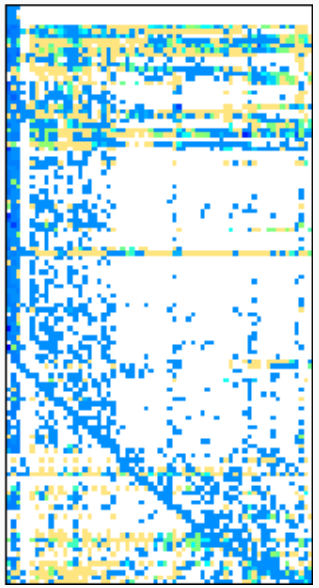
Many applications from physics, engineering, chemistry, geodesy, etc, require the solution of a linear system like

$Ax = b$, with A, rectangular, sparse and potentially large

$$\begin{array}{llllll} m \geq n & \min_x \|Ax - b\|_2 & \rightarrow & QR = A, & z = Q^T b, & x = R^{-1} z \\ m < n & min\|x\|_2, \quad Ax = b & \rightarrow & QR = A^T, & z = R^{-T} b, & x = Qz \end{array}$$

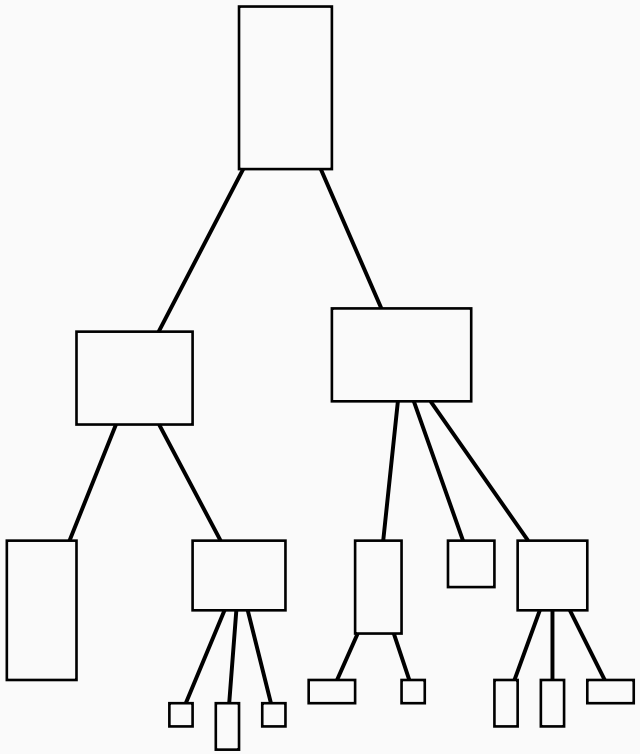
A sparse matrix is mostly filled with zeros:

- Reduce memory storage.
- Reduce computational costs.
- Generate parallelism.



THE MULTIFRONTAL (QR) FACTORIZATION

The Multifrontal method in a nutshell



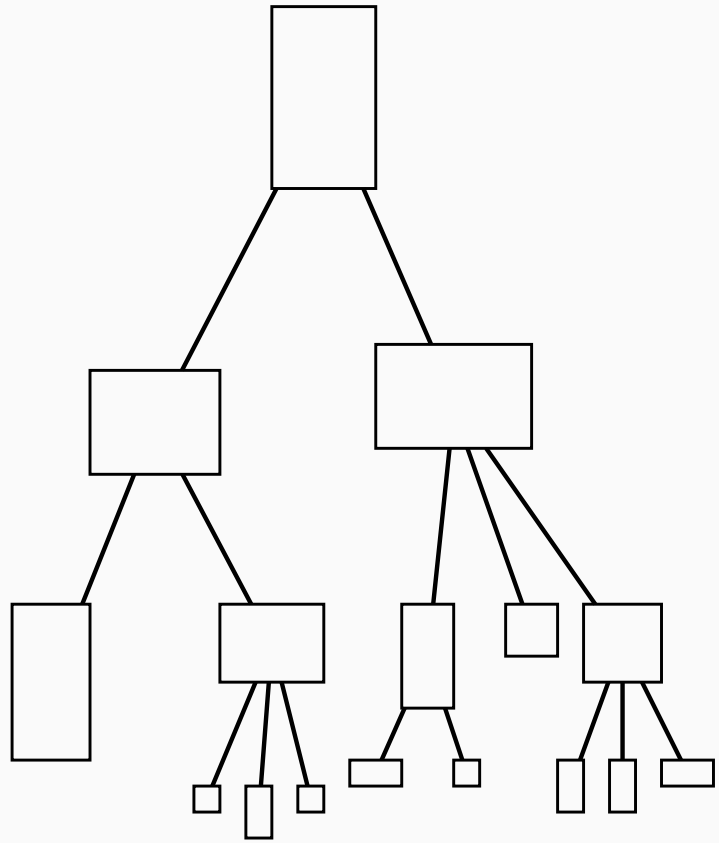
```
forall fronts f in topological order

   ! allocate and initialize front
   call activate(f)

   ! front assembly
   forall children c of f
      call assemble(c, f)
      ! Deactivate child
      call deactivate(c)
   end do

   ! front factorization
   call factorize(f)
end do
```

- The elimination tree is pre-computed based on the matrix structure
- Nodes are dense matrices called fronts which can be very small ($O(1)$) or very large ($O(10^5)$), square or rectangular
- In the actual factorization the tree is traversed bottom up and when a node is visited, the corresponding front is allocated, assembled (using children) and factorized

The Multifrontal QR method

The original multifrontal method by Duff & Reid '83 can be extended to QR factorization of sparse matrices.
This method is guided by a graph called *elimination tree*:

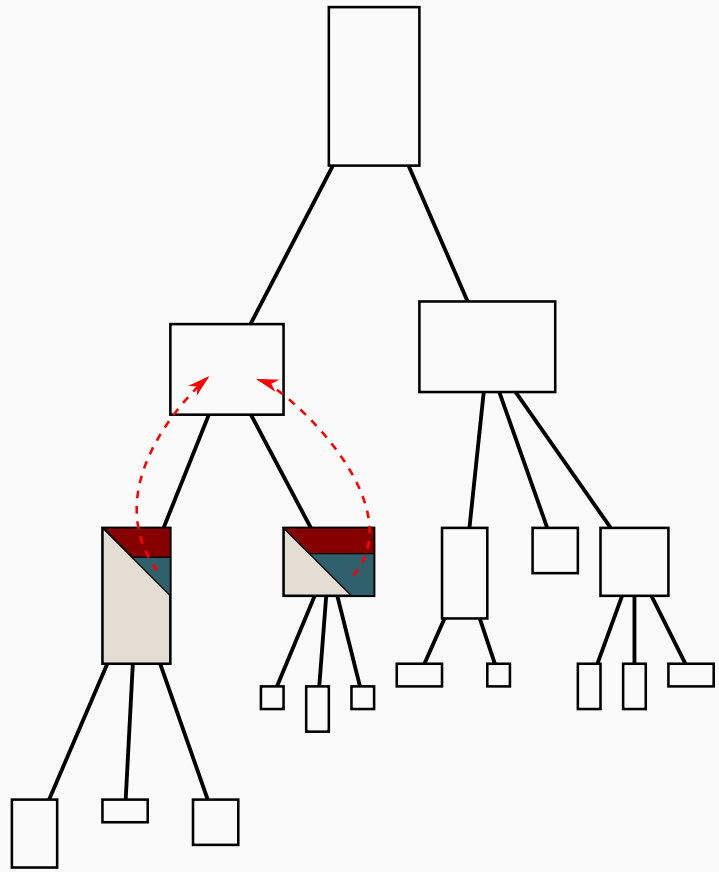
- each node is associated with a relatively small dense matrix called frontal matrix (or front) containing k pivots to be eliminated along with all the other coefficients concerned by their elimination.



The Multifrontal QR method

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

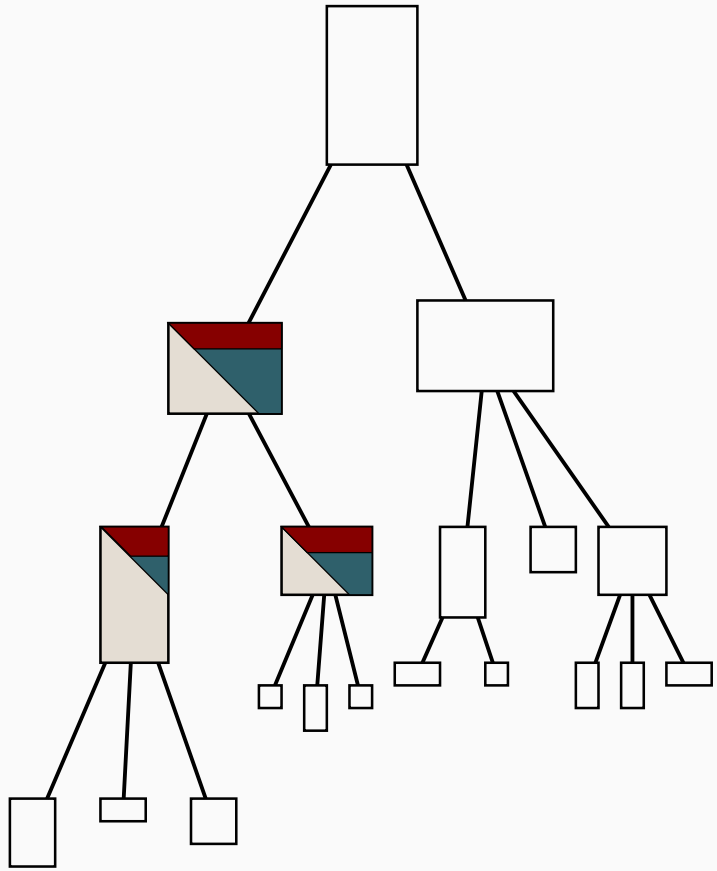
- assembly: coefficients from the original matrix associated with the pivots and *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix.



The Multifrontal QR method

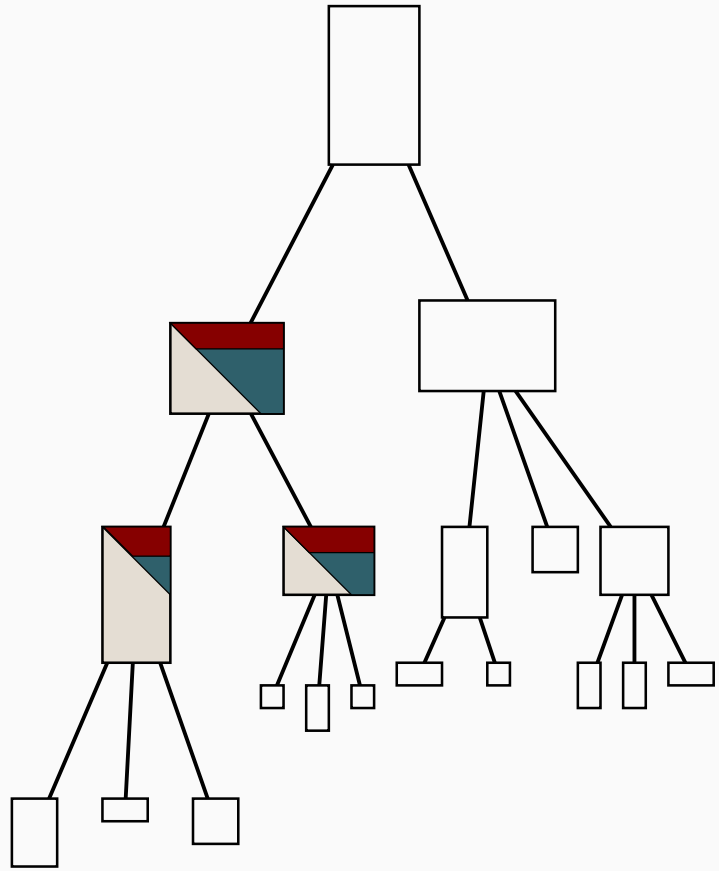
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- assembly: coefficients from the original matrix associated with the pivots and *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 dense QR factorization of the frontal matrix. As a result we get:
  - part of the global R and Q factors.
  - a triangular *contribution block* that will be assembled into the parent's front.



The Multifrontal QR method

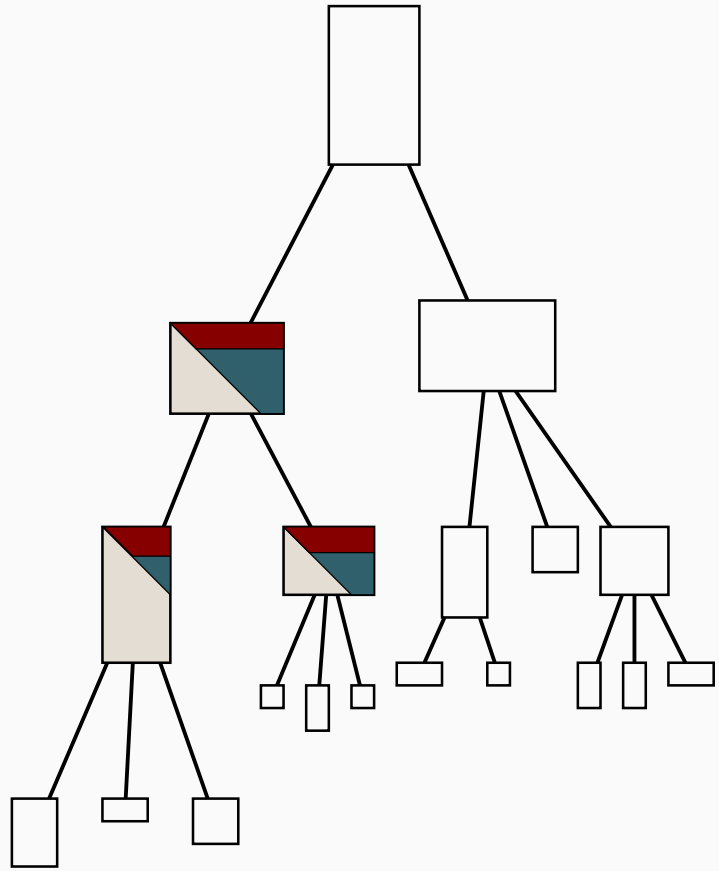
Typically two sources of parallelism are exploited in the multifrontal method



The Multifrontal QR method

Typically two sources of parallelism are exploited in the multifrontal method

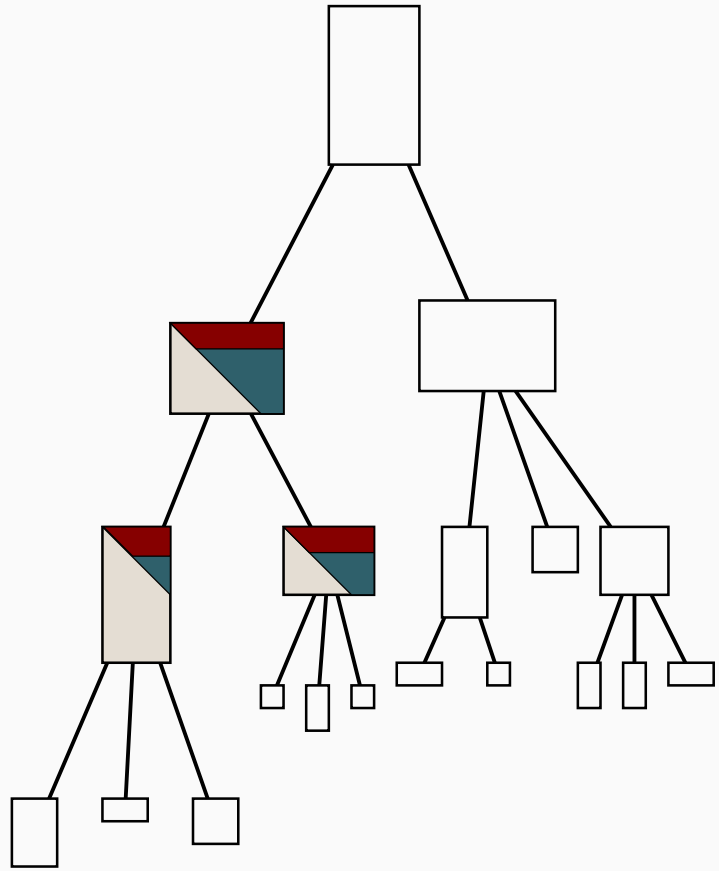
- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.



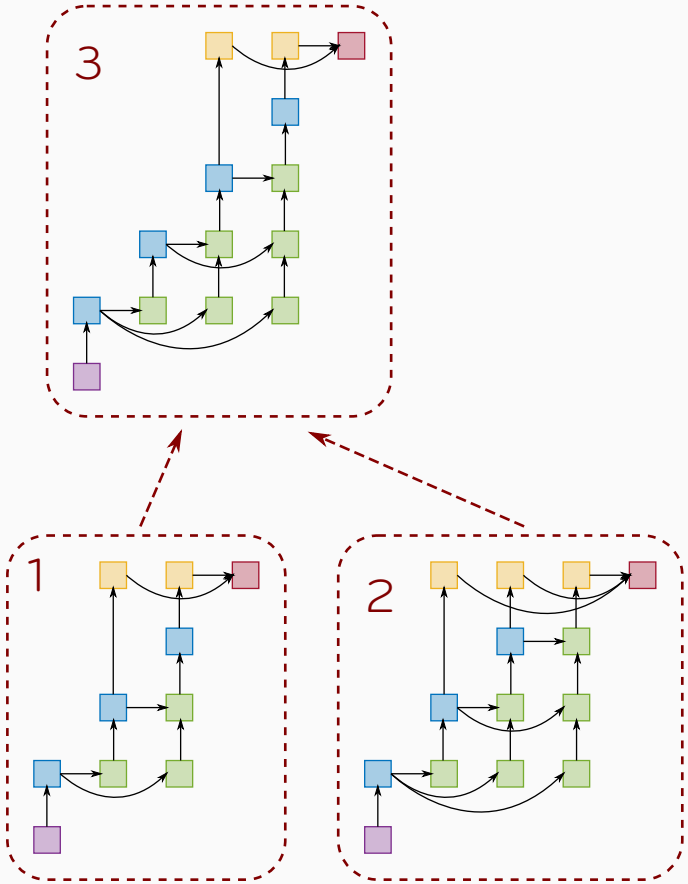
The Multifrontal QR method

Typically two sources of parallelism are exploited in the multifrontal method

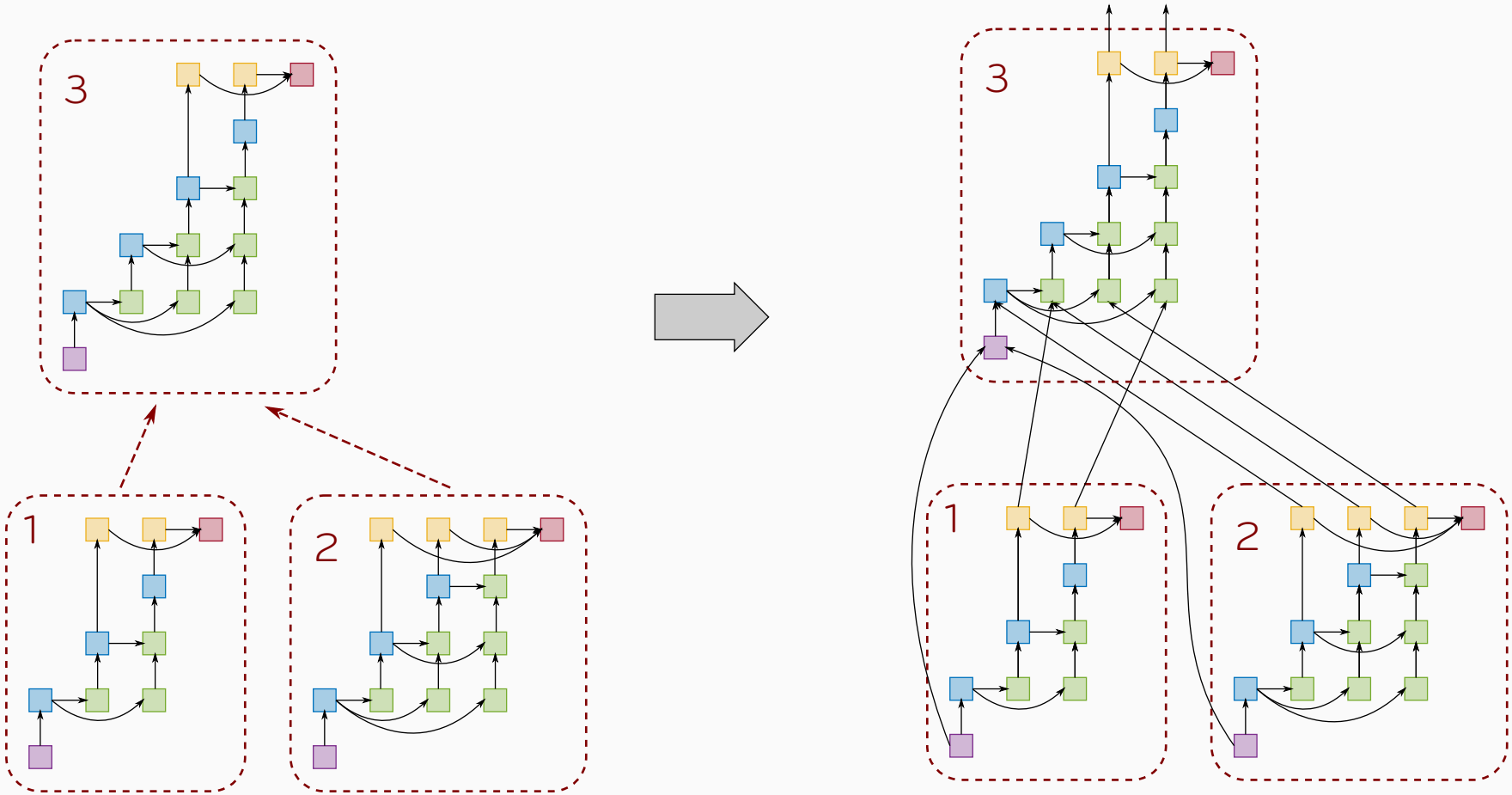
- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.
- node-level parallelism: large frontal matrices factorization may be performed in parallel by multiple threads.



Improved tree parallelism



Improved tree parallelism



Finer tracking of the dependencies between operations allows for pipelining the processing of a node with that of its children. We refer to this extra source of concurrency as *Inter-Level Parallelism*

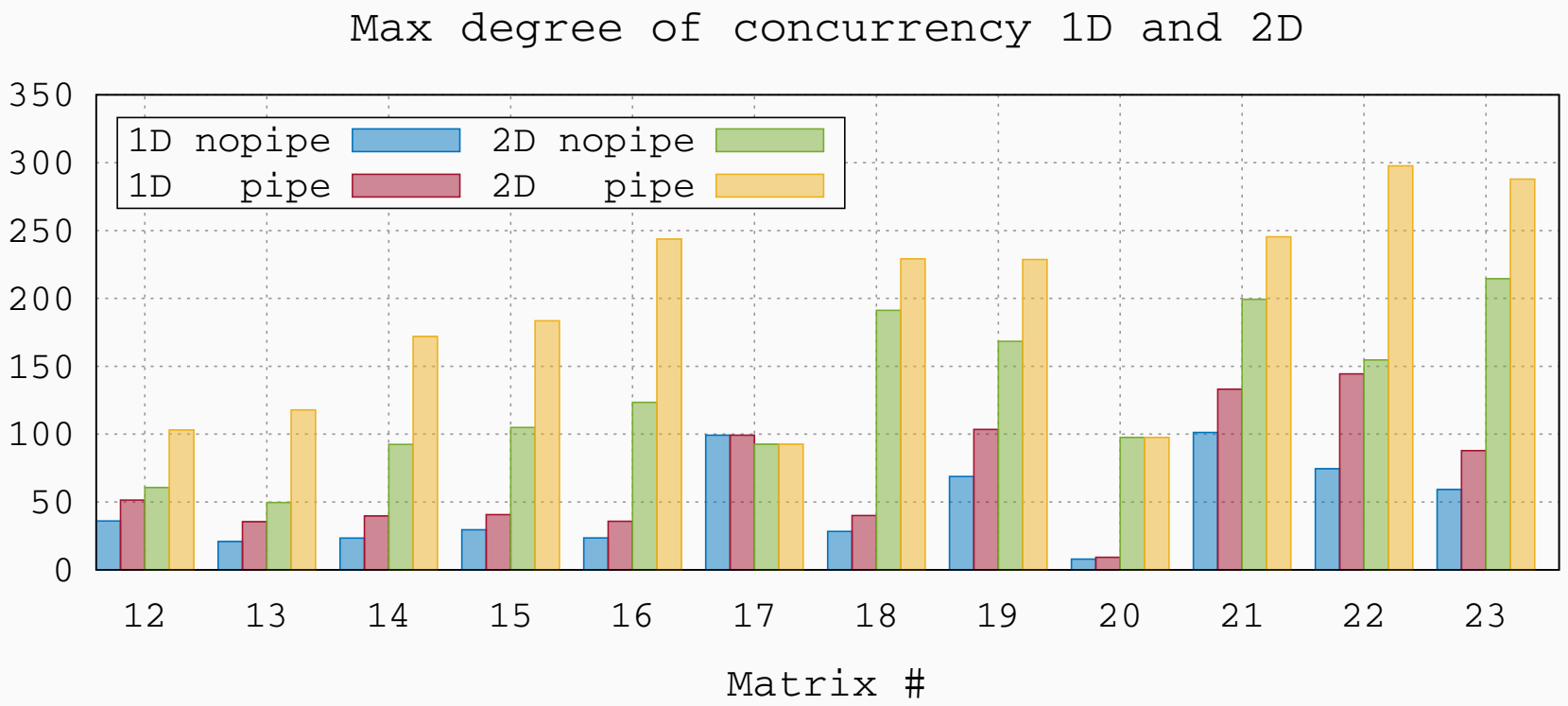
Improved parallelism

Matrices from the Suite Sparse Matrix Collection

#	Mat. name	m	n	nz	op. count
12	hirlam	1385K	452K	2713K	1384G
13	flower_8_4	55K	125K	375K	2851G
14	Rucci1	1977K	109K	7791K	5671G
15	ch8-8-b3	117K	18K	470K	10709G
16	GL7d24	21K	105K	593K	16467G
17	neos2	132K	134K	685K	20170G
18	spal_004	10K	321K	46168K	30335G
19	n4c6-b6	104K	51K	728K	62245G
20	sls	1748K	62K	6804K	65607G
21	TF18	95K	123K	1597K	194472G
22	lp_nug30	95K	123K	1597K	221644G
23	mk13-b5	135K	270K	810K	259751G

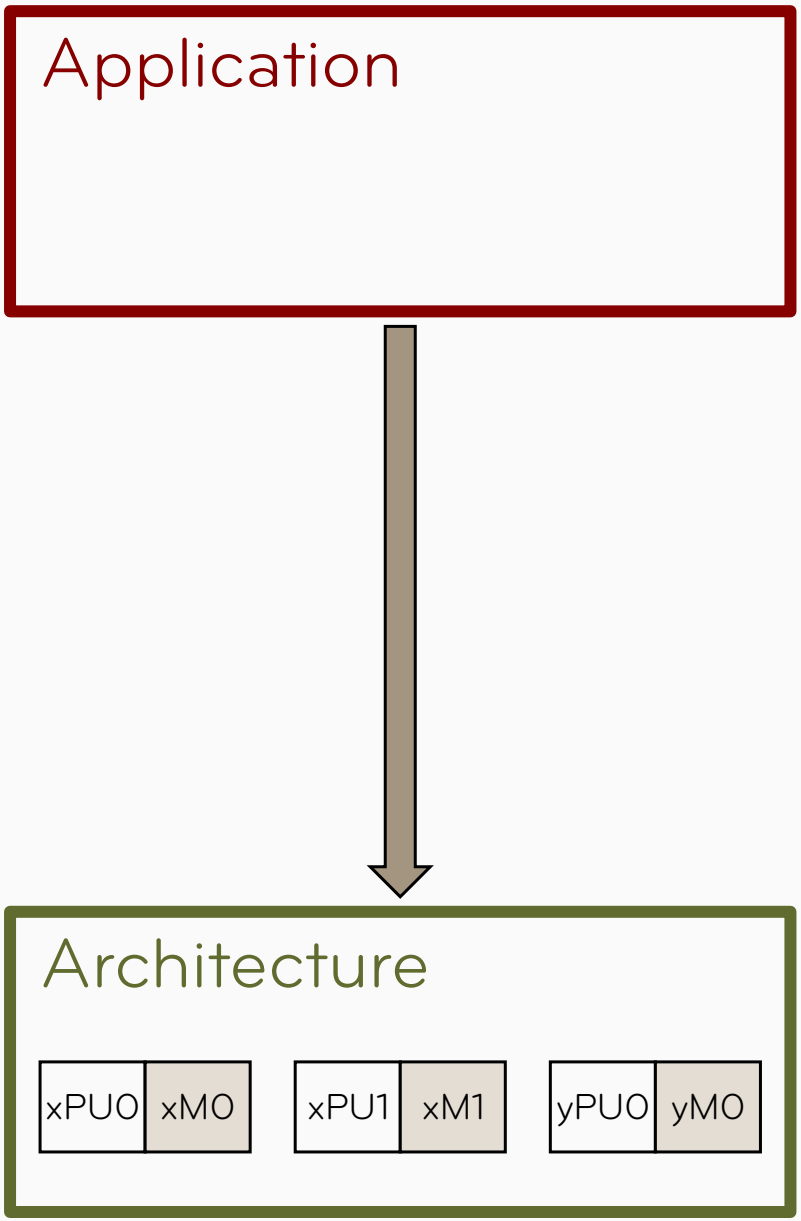
Improved parallelism

$$\text{concurrency} = \frac{\text{sequential time}}{\text{Critical Path}}$$



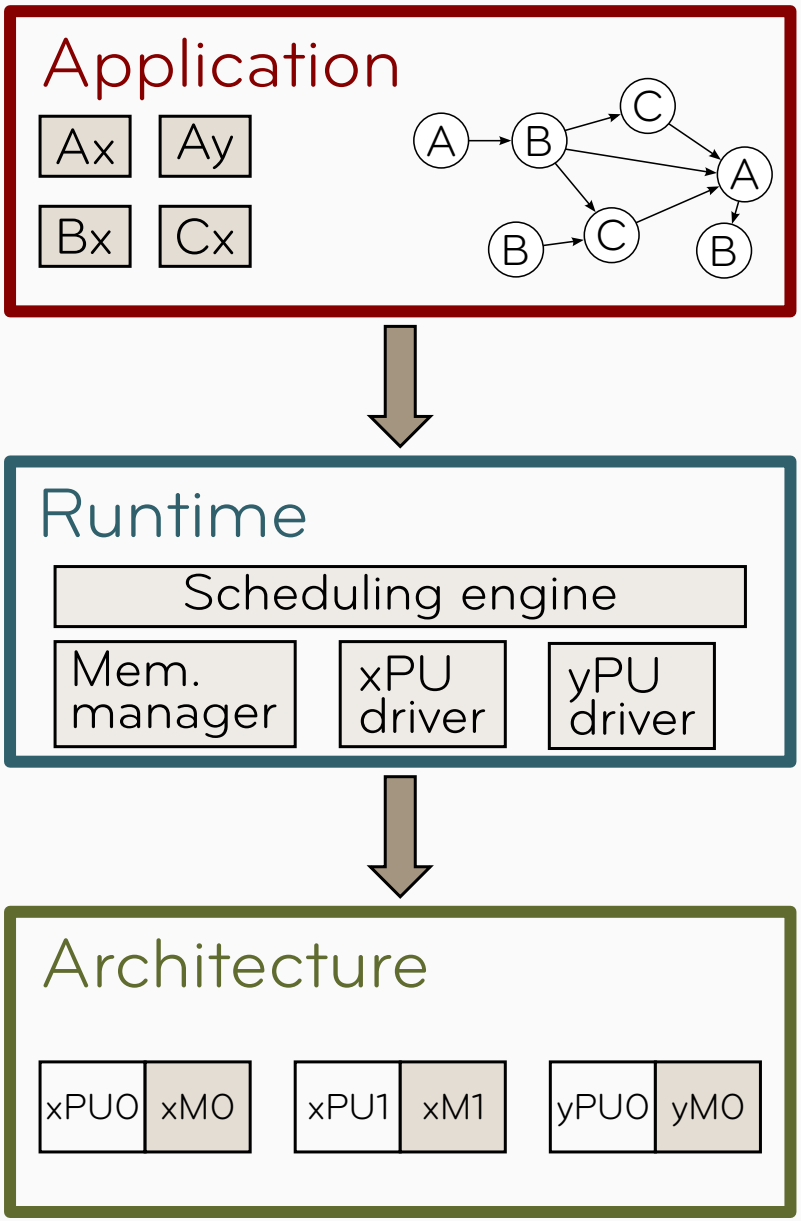
TASK PARALLELISM AND RUNTIME SYSTEMS

Runtime systems



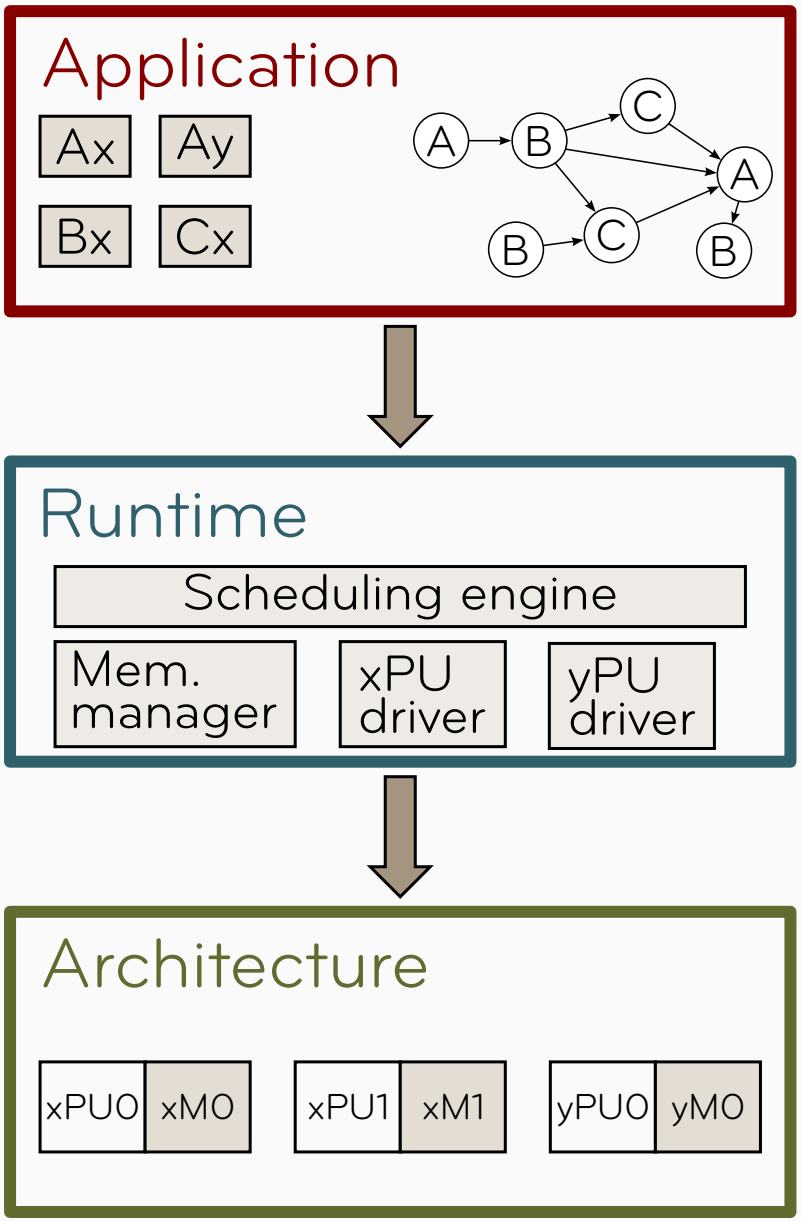
- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.

Runtime systems



- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.
- runtimes provide an abstraction layer that hides the architecture details.

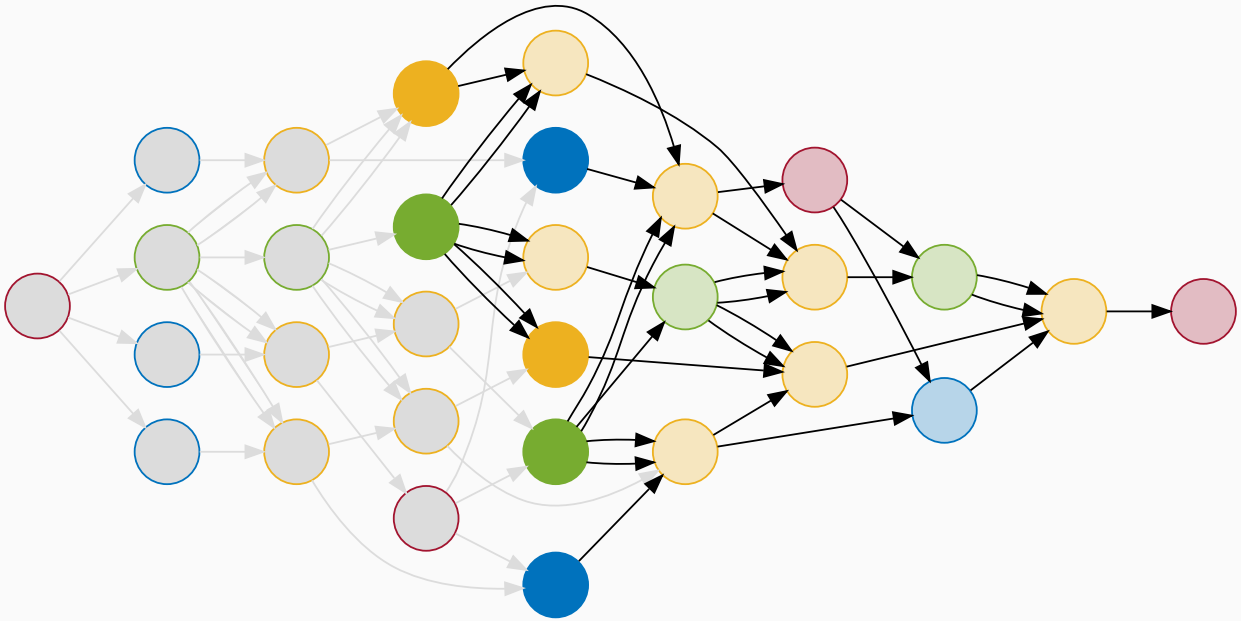
Runtime systems



- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.
- runtimes provide an abstraction layer that hides the architecture details.
- the workload is expressed as a DAG (Directed Acyclic Graph) of tasks.

Task parallelism

Workload is expressed as a Directed Acyclic Graph of tasks



- Inherently expresses concurrency and data flow
- Asynchronous execution
- Allows for dynamic scheduling policies

Sequential Task Flow runtimes

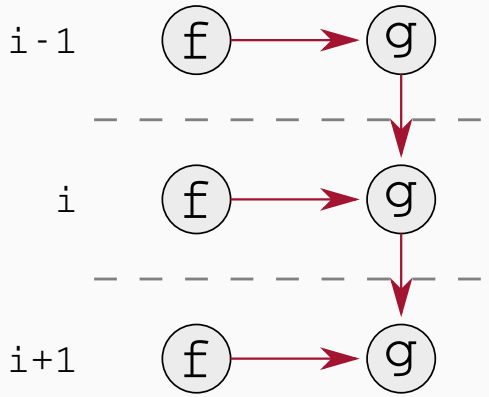
The Sequential Task Flow is a portable programming model where tasks are submitted sequentially and their execution is delegated to a runtime

```
for (i = 1; i < N; i++) {
  x[i] = f(x[i]);
  y[i] = g(x[i], y[i-1]);
}
```

Sequential Task Flow runtimes

The Sequential Task Flow is a portable programming model where tasks are submitted sequentially and their execution is delegated to a runtime

```
for (i = 1; i < N; i++) {
  submit(f, x[i]:RW);
  submit(g, y[i]:R, x[i]:R, y[i-1]:R)
      ;
}
```



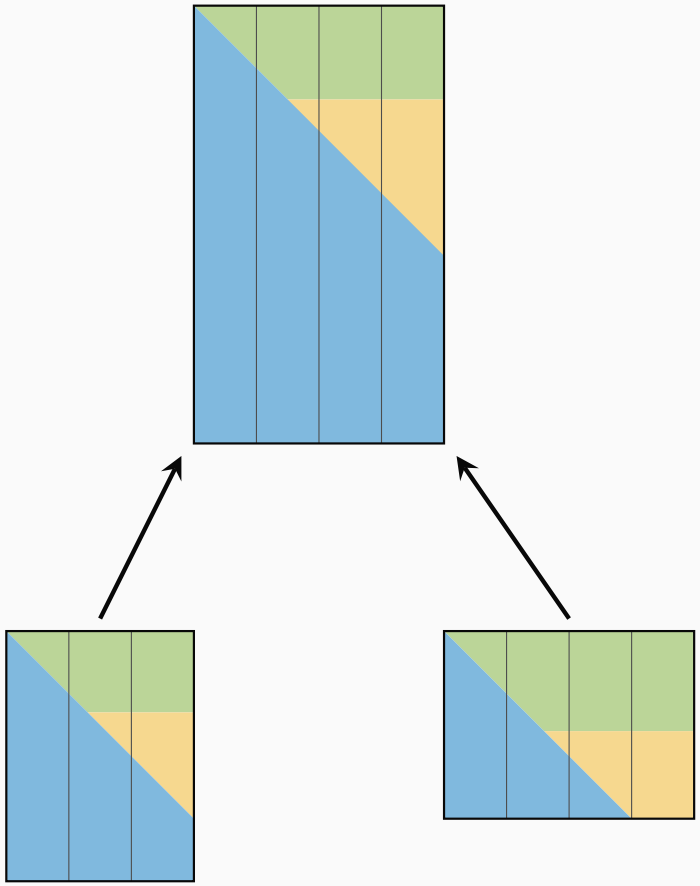
Dependencies are automatically computed through data analysis

We have chosen StarPU because of its numerous features

- data management/transfer
- plug&play scheduling policies
- support for accelerators
- support for distributed memory
- ...

STF MULTIFRONTAL QR

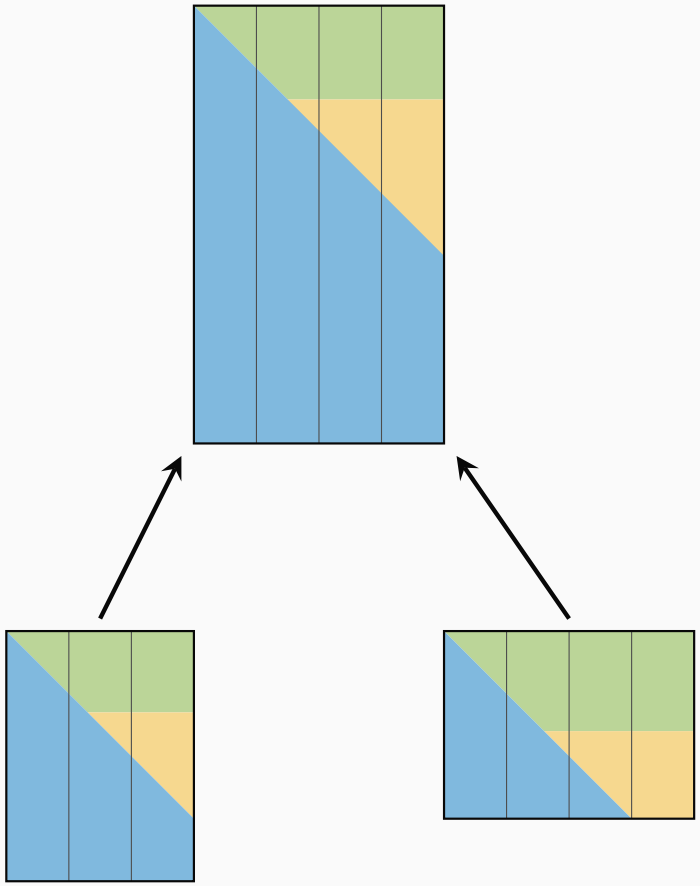
The task-based multifrontal QR factorization



```
do f=1, nfronts ! in postorder
   ! compute front structure
   call activate(f)
   ! allocate and initialize front
   call init(f)

   do c=1, f%nc ! for all the children of f
      do j=1,c%n
         ! assemble column j of c into f
         call assemble(c(j), f)
      end do
      ! Deactivate child
      call deactivate(c)
   end do

   do p=1, f%n
      ! panel reduction of column p
      call _geqrt(f(p))
      do u=p+1, f%n
         ! update of column u with panel p
         call _gemqrt(f(p), f(u))
      end do
   end do
end do
```

Sequential multifrontal *QR* code with 1D block partitioning

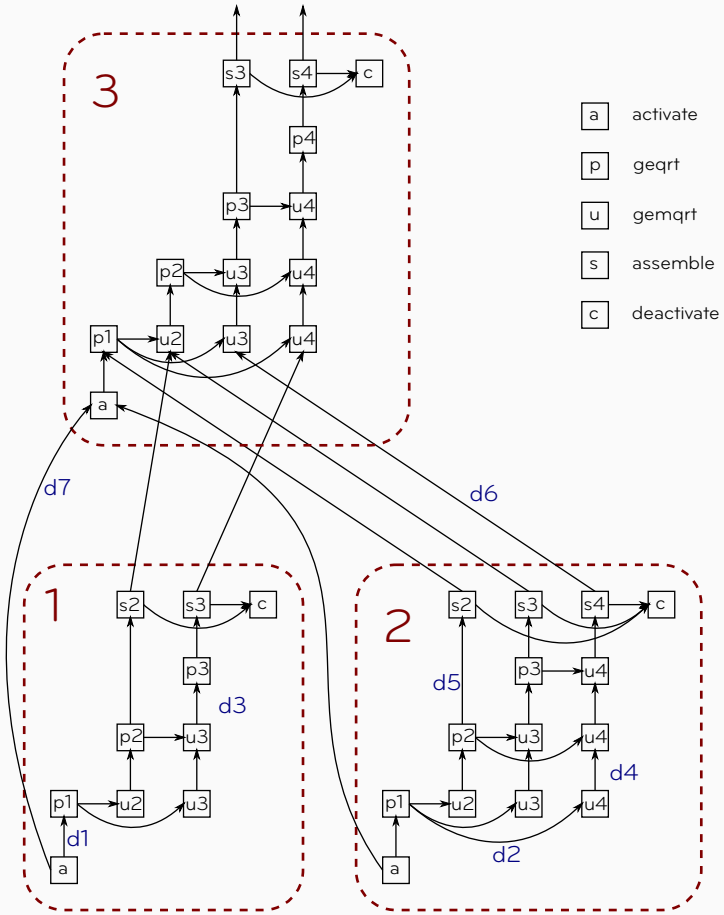
The task-based multifrontal QR factorization



```
do f=1, nfronts ! in postorder
   ! compute structure and register handles
   call activate(f)
   ! allocate and initialize front
   call submit(init, f:RW)

   do c=1, f%nc ! for all the children of f
      do j=1,c%n
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   do p=1, f%n
      ! panel reduction of column p
      call submit(_geqrt, f(p):RW)
      do u=p+1, f%n
         ! update of column u with panel p
         call submit(_gemqrt, f(p):R, f(u):RW)
      end do
   end do
end do
! wait for the tasks to be executed
call wait_tasks_completion()
```

- STF multifrontal *QR* code with 1D block partitioning
- Elimination tree is transformed into a DAG

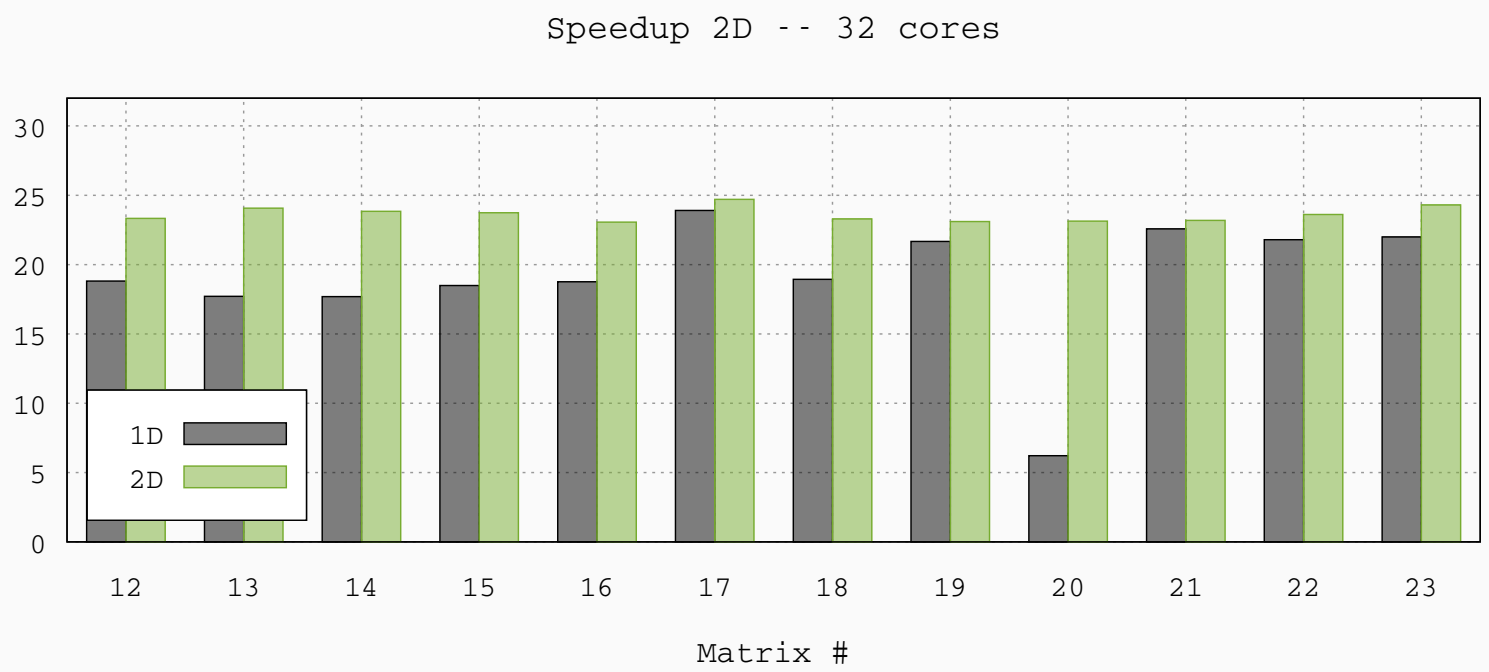
The task-based multifrontal QR factorization




```
do f=1, nfronts ! in postorder
   ! compute structure and register handles
   call activate(f)
   ! allocate and initialize front
   call submit(init, f:RW)

   do c=1, f%nc ! for all the children of f
      do j=1,c%n
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   do p=1, f%n
      ! panel reduction of column p
      call submit(_geqrt, f(p):RW)
      do u=p+1, f%n
         ! update of column u with panel p
         call submit(_gemqrt, f(p):R, f(u):RW)
      end do
   end do
end do
! wait for the tasks to be executed
call wait_tasks_completion()
```

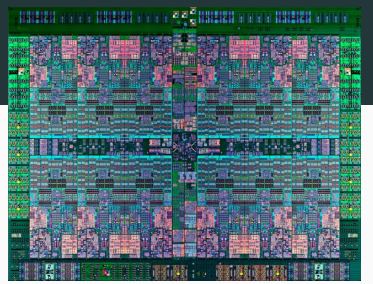
- Seamless exploitation of tree and node parallelism.
- Inter-level concurrency (parent-child pipelining).

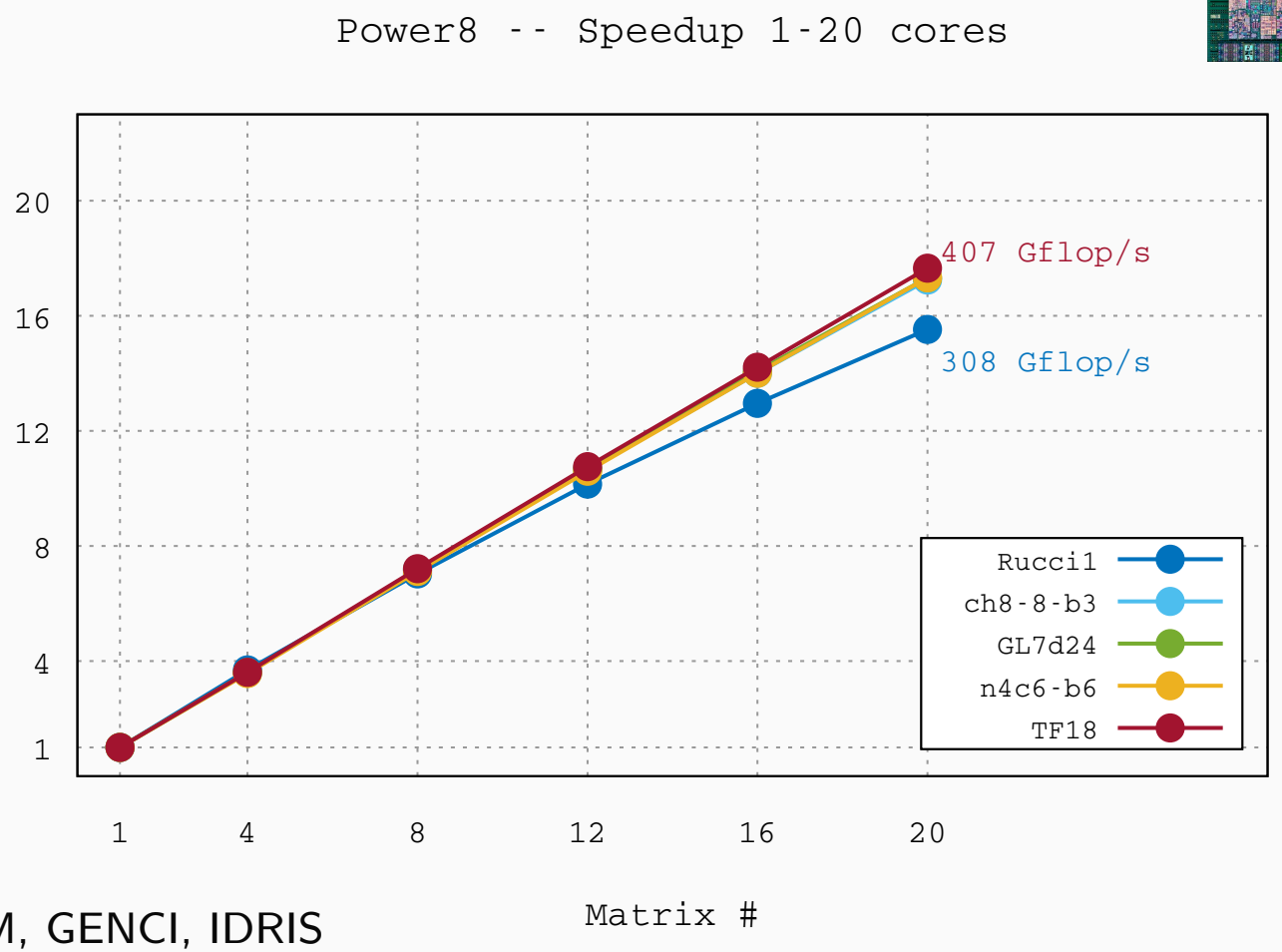
Experimental results: speedups



- Improved node and tree parallelism bring great benefit to small size and strongly overdetermined problems
- Speedups are uniform for all tested matrices.
- Performance ranges from 321 to 462 Gflop/s (46% to 66% of the peak)

More experimental results

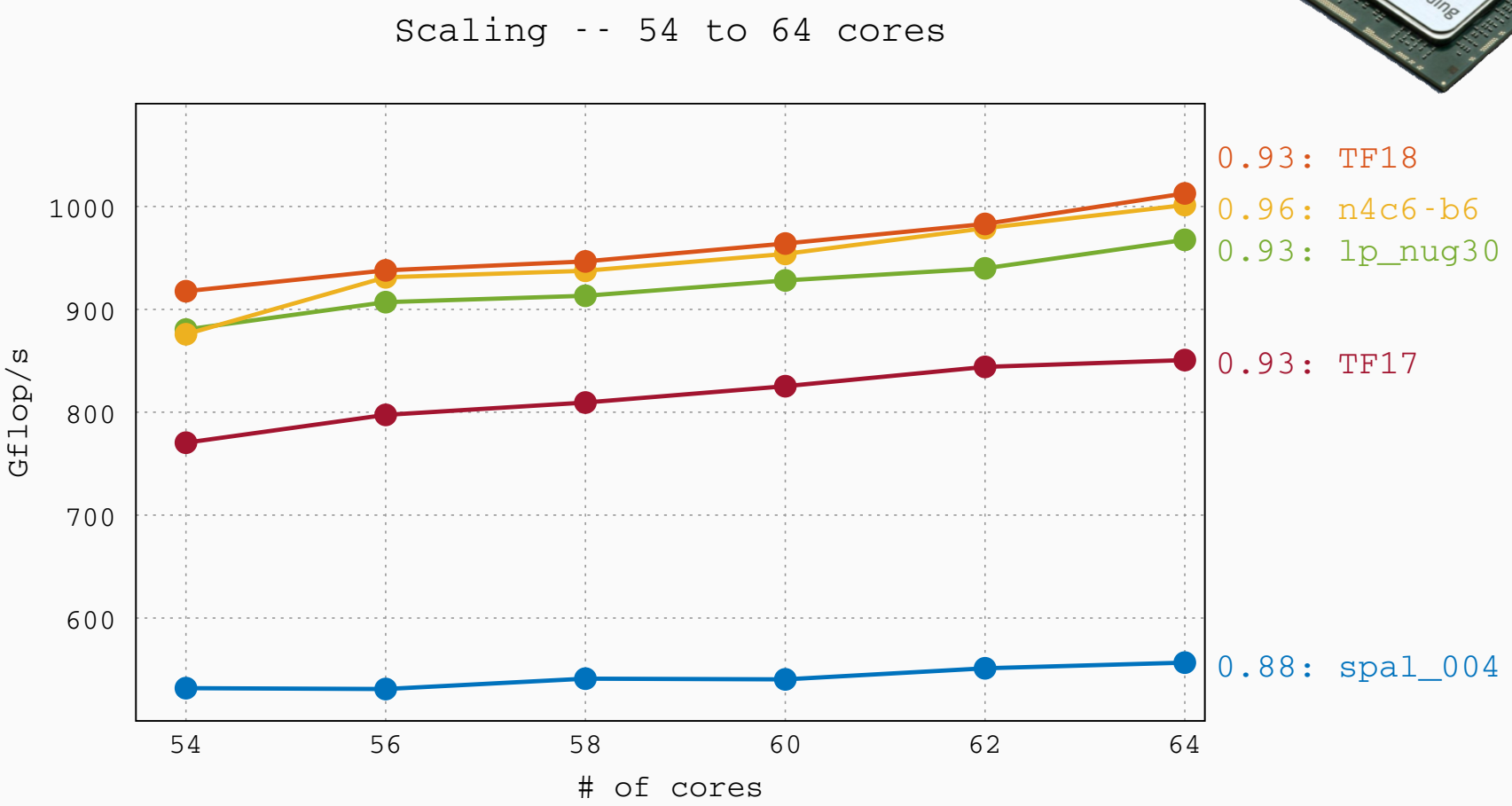




Credits: IBM, GENCI, IDRIS

On a 2 x Power8 machine 88% of parallel efficiency on 20 cores

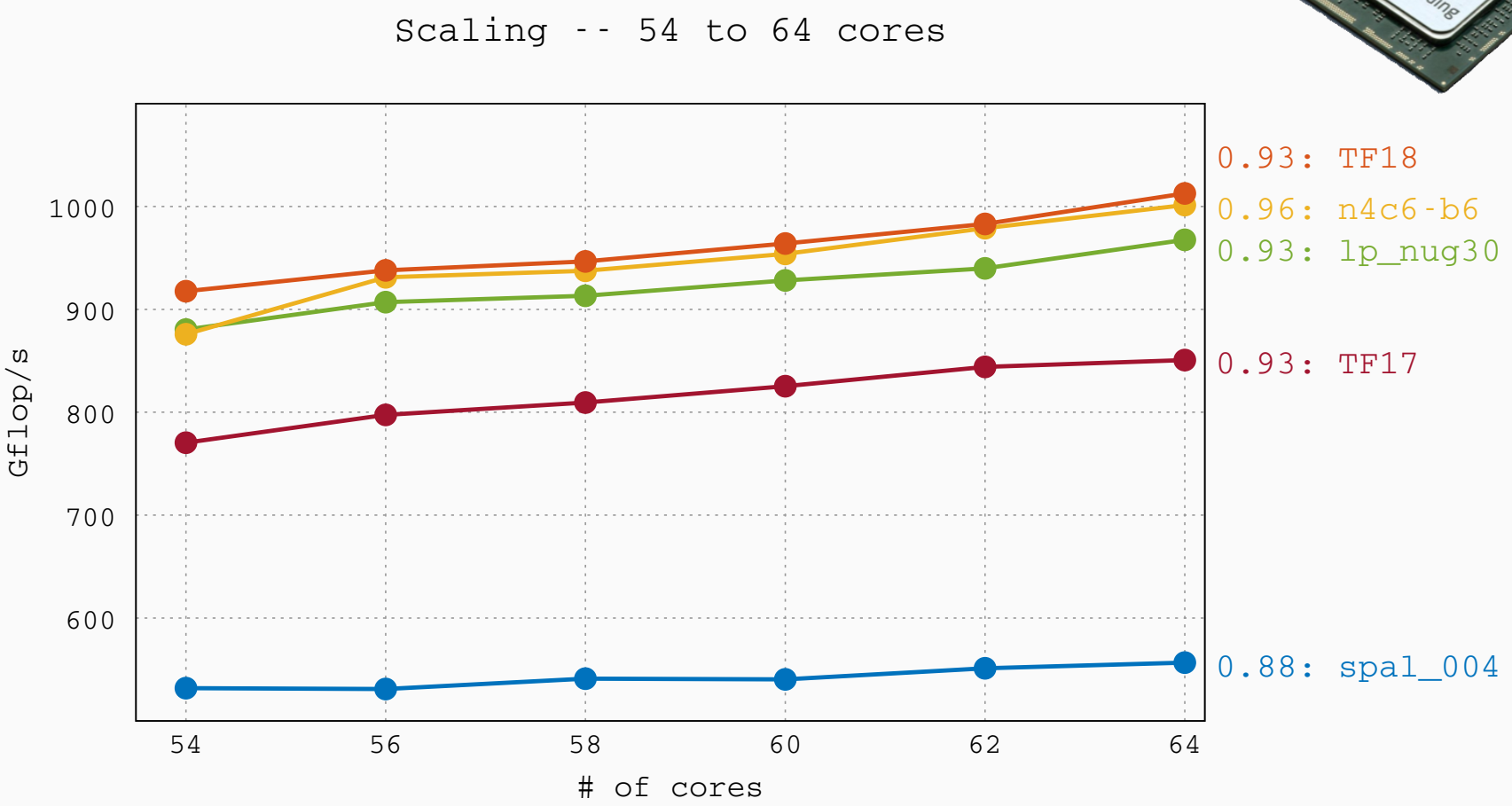
Intel Knights Landing



Credits: PlaFRIM

- Cores in quadrant mode
- MCDRAM in cache mode
- 8000 pages of 2MB and THP on
- Scalable allocator from TBB is used

Intel Knights Landing



Credits: PlaFRIM

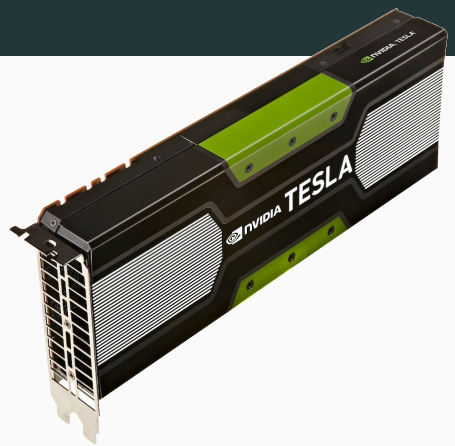
Efficiency is computed as

$$e = \frac{t_{54}}{t_{64}} / \frac{64}{54}$$

STF-BASED PARALLEL MULTIFRONTAL QR METHOD FOR HETEROGENEOUS ARCHITECTURES

GPU-based systems

- Very high computing power ($O(1)$ Tflop/s)
- Very high memory bandwidth ($O(100)$ GB/s)
- Very convenient Gflops/s/Watt ratio ($O(10)$)



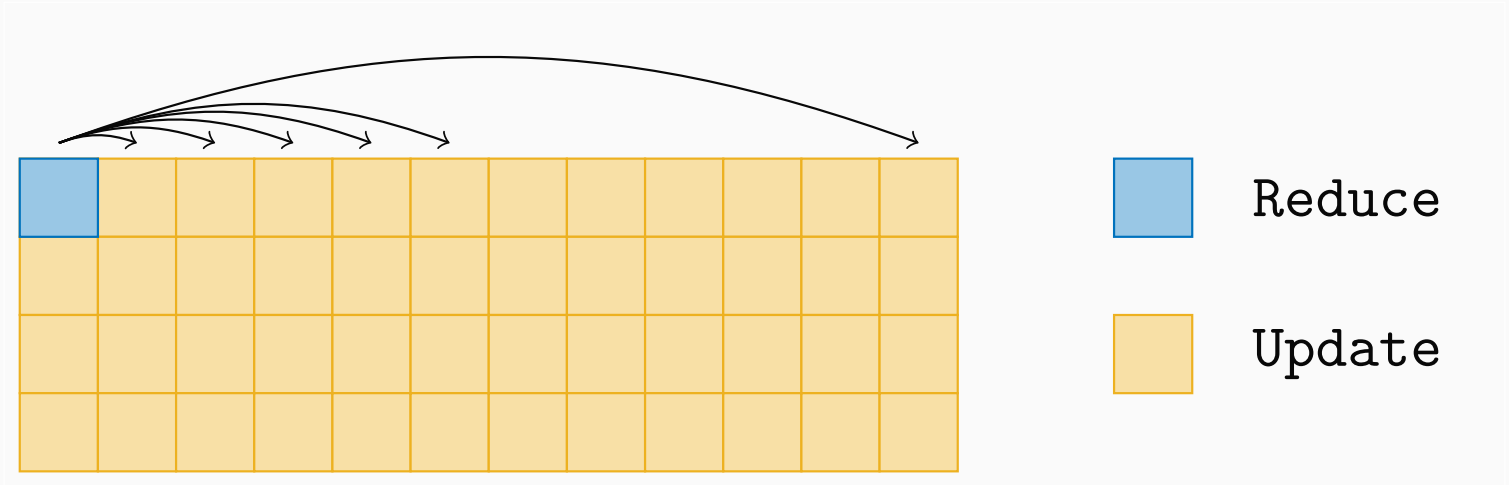
Objective

Exploit heterogeneity (i.e. take advantage of the diversity of resources) to accelerate the multifrontal QR factorization.

Issues:

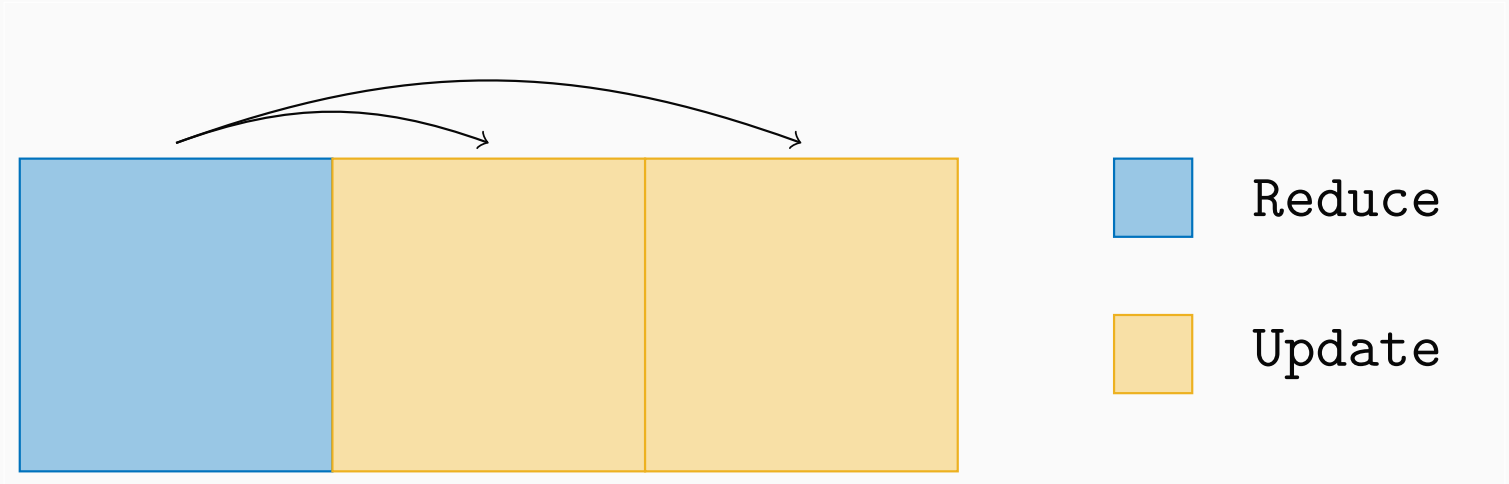
- Granularity: GPUs require coarser grained tasks to achieve full speed;
- Scheduling: account for different computing capabilities and different tasks characteristics while maximizing concurrency;
- Communications: minimize the cost of host-to-device data transfers.

Frontal matrices partitioning strategies



- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU

Frontal matrices partitioning strategies

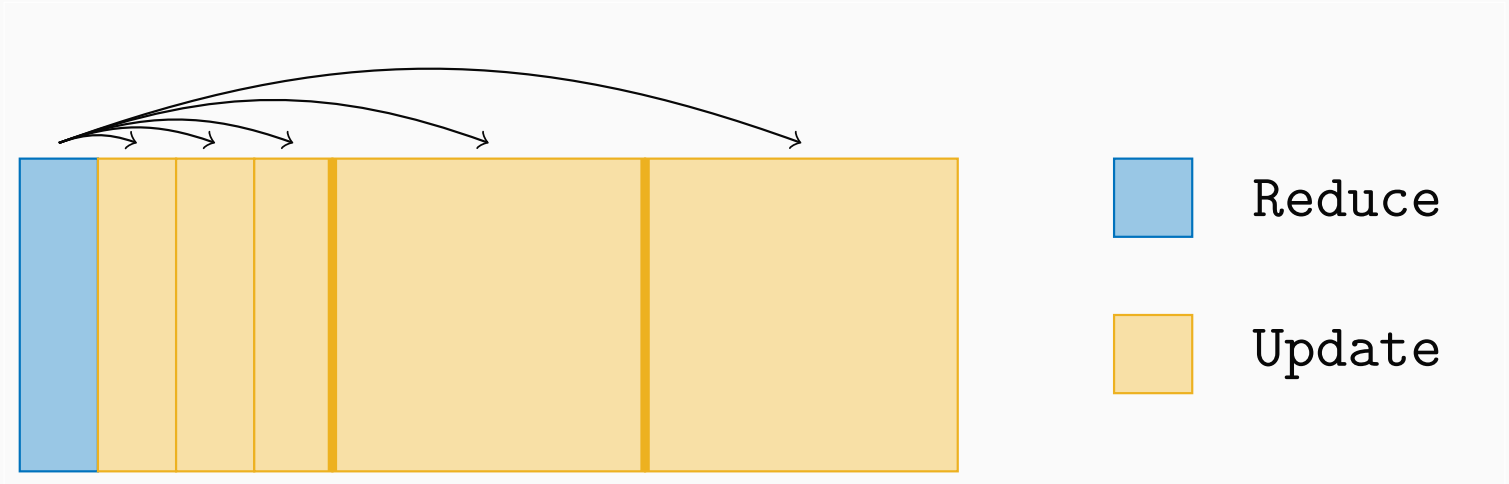


- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency

Frontal matrices partitioning strategies

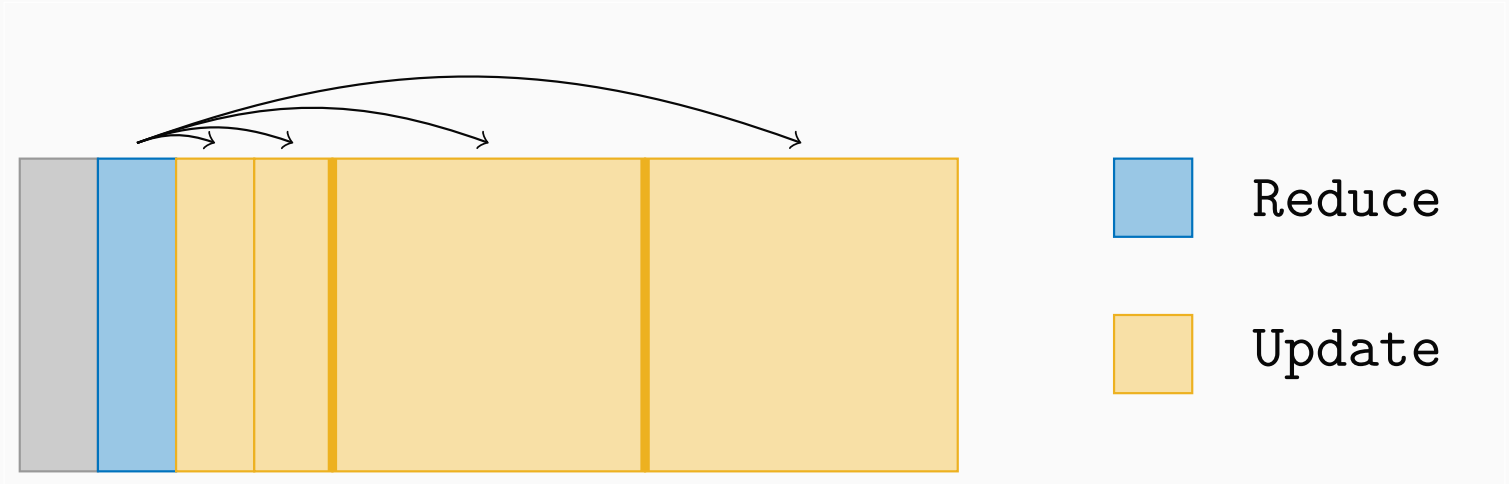
- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

Frontal matrices partitioning strategies



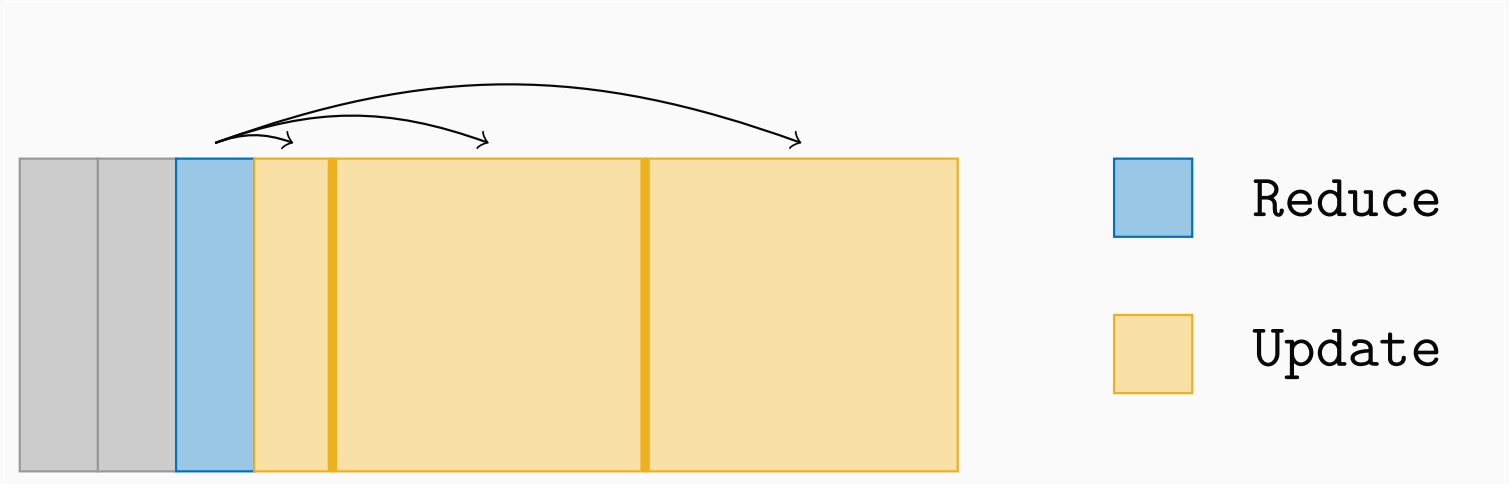
- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

Frontal matrices partitioning strategies



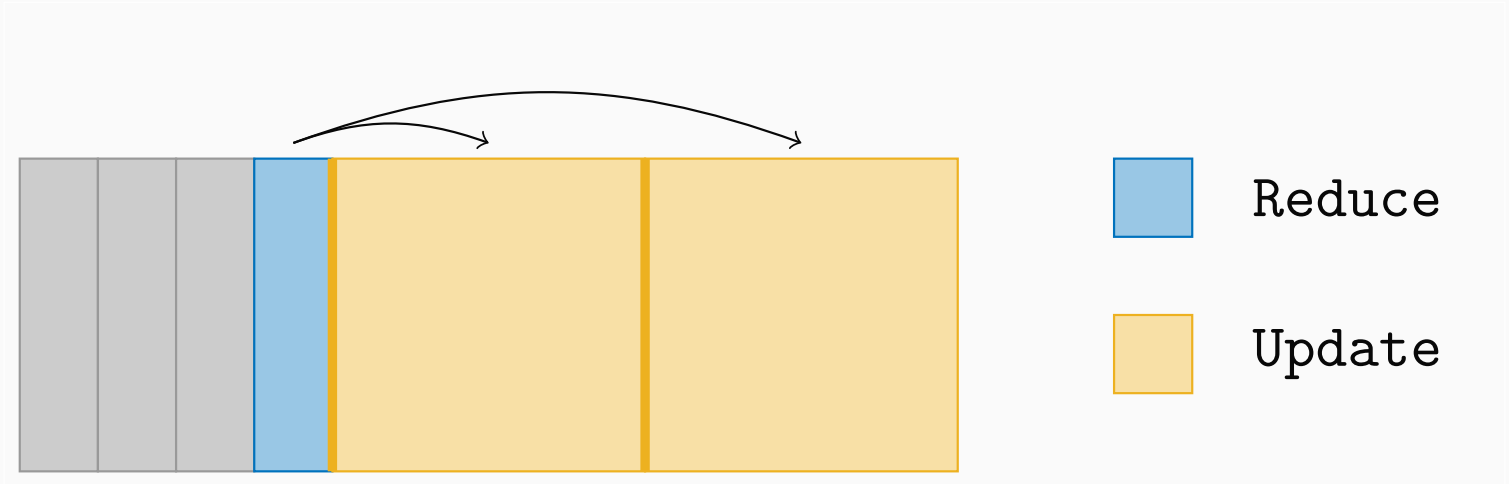
- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

Frontal matrices partitioning strategies



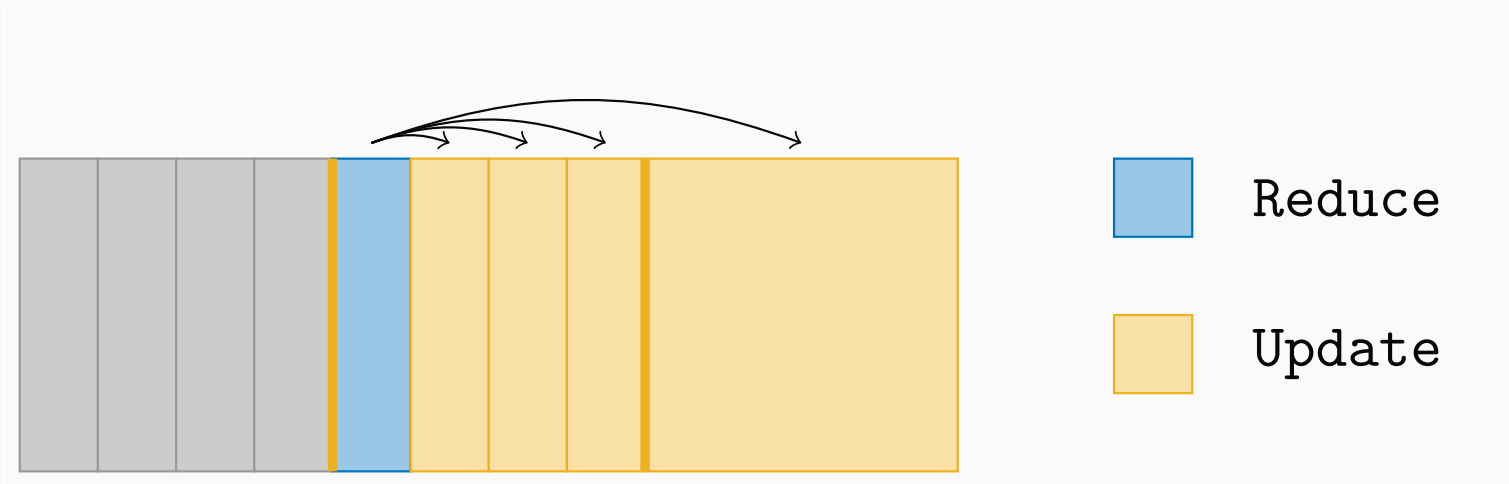
- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

Frontal matrices partitioning strategies



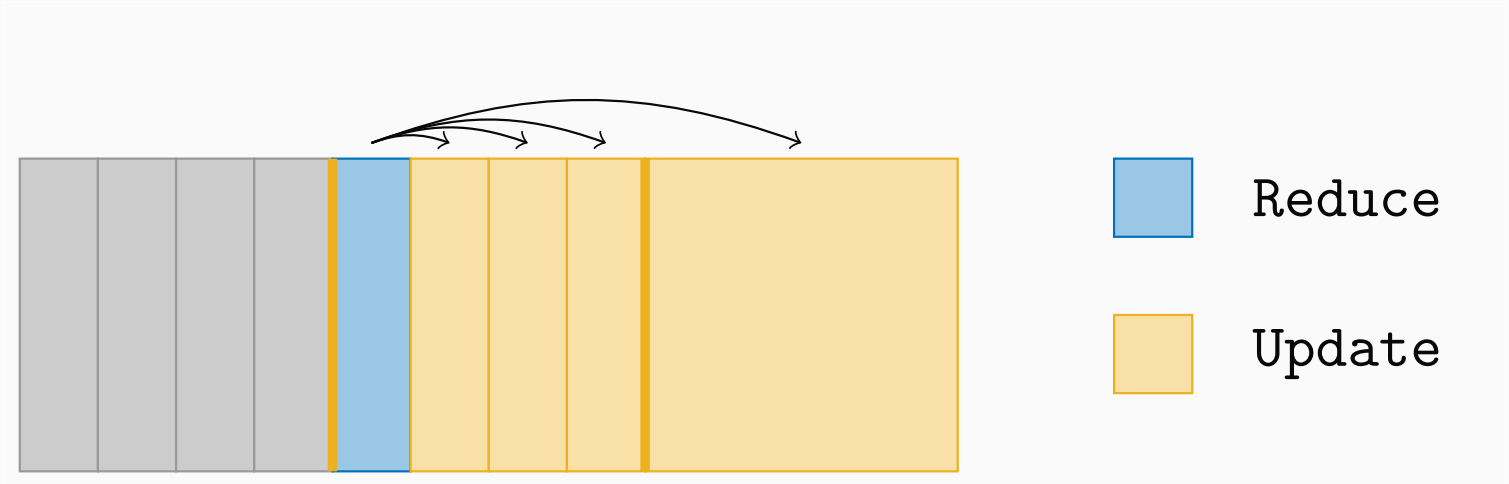
- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

Frontal matrices partitioning strategies



- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

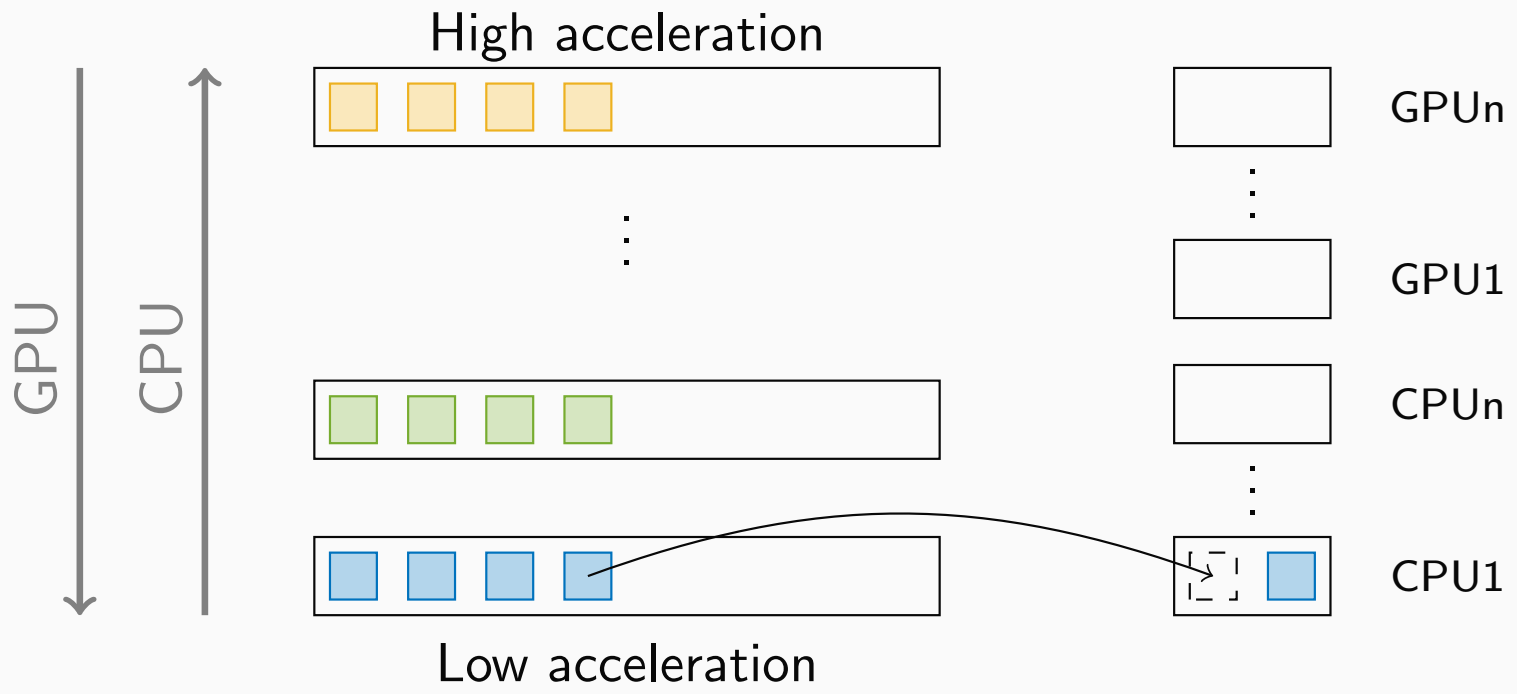
Frontal matrices partitioning strategies



- **Fine grain** partitioning provides high concurrency but low tasks efficiency on GPU
- **Coarse grain** partitioning achieves optimum granularity for GPU but limited concurrency
- Hierarchical, dynamic partitioning
  - granularity and concurrency trade-off.
  - heterogeneity to be exploited.

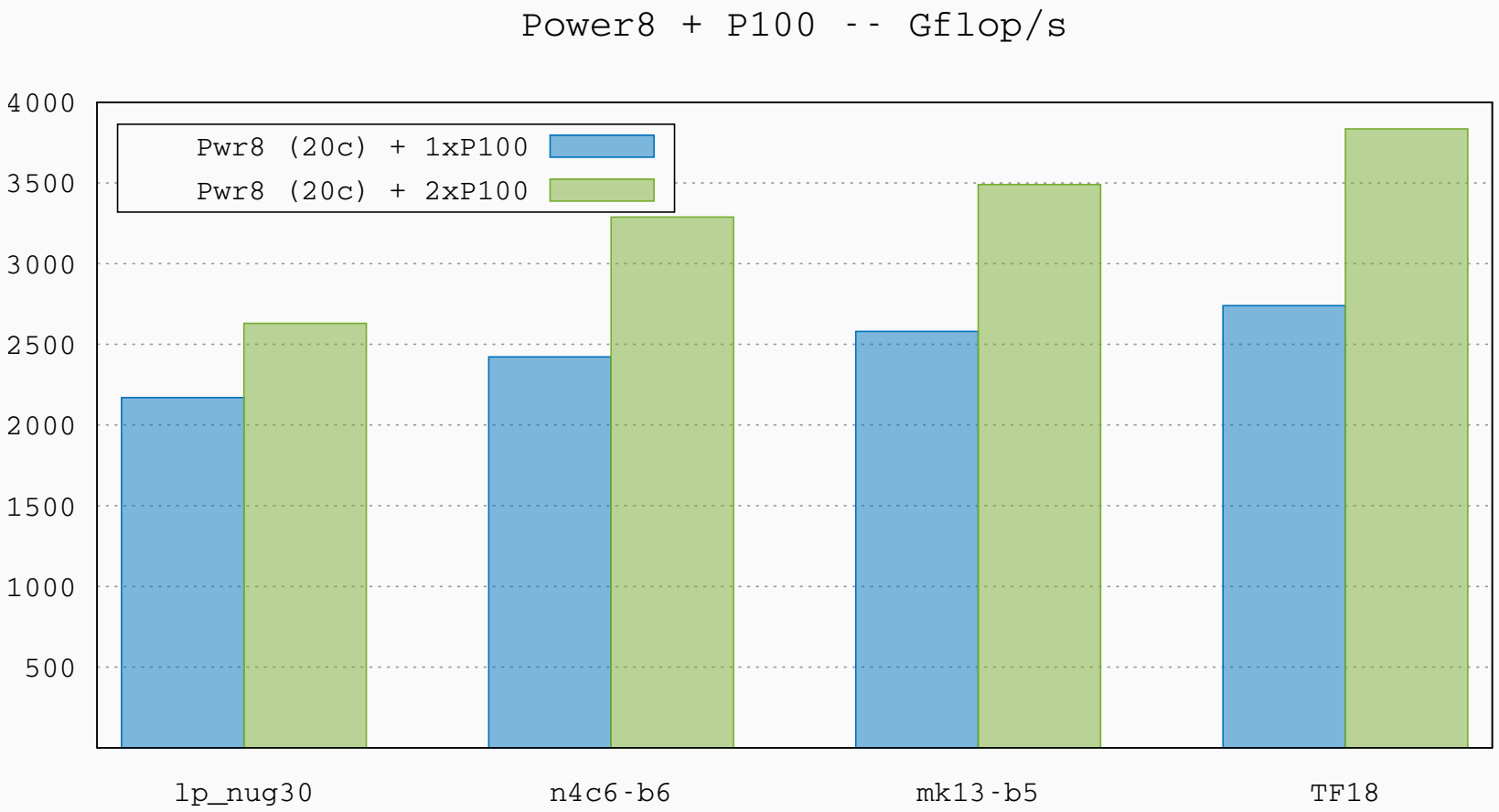
The dynamic (un)partitioning of frontal matrices is achieved through dedicated tasks $\rightarrow$ StarPU handles the consistency among partitions.

HeteroPrio scheduling policy



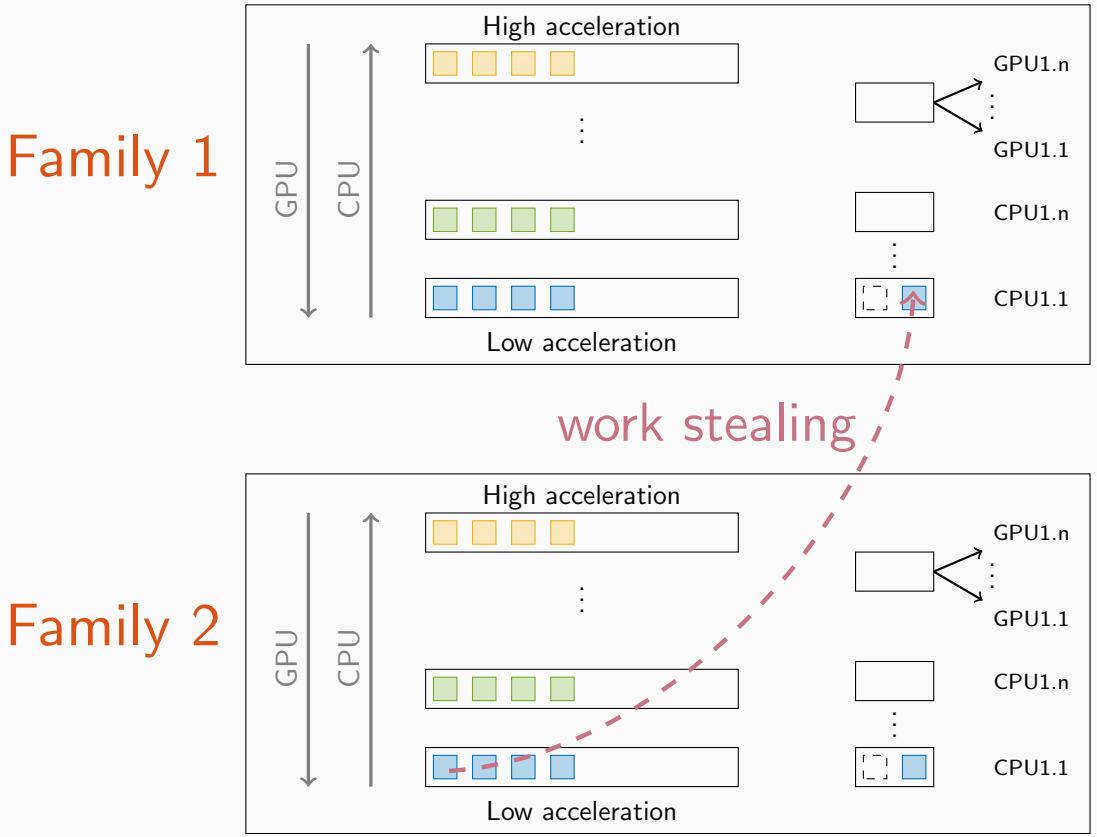
- Ready tasks are sorted in different queues/buckets depending on the acceleration factor (ratio between GPU and CPU performance)
- Buckets are traversed in different order by CPUs and GPUs when looking for a ready task
- Upon selection, a ready task is moved to the corresponding Worker Queue and its data prefetched

Experimental results



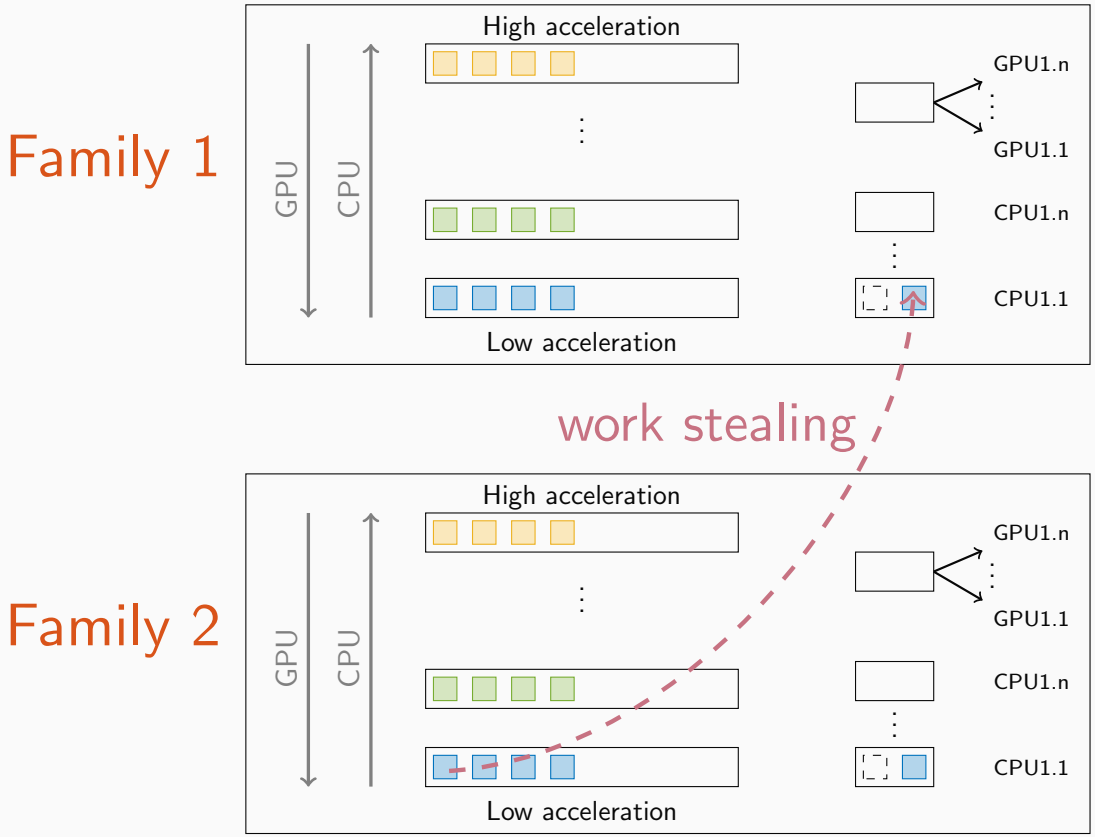
Thanks to IDRIS, PlaFRIM, CALMIP and GENCI for providing access to the resources

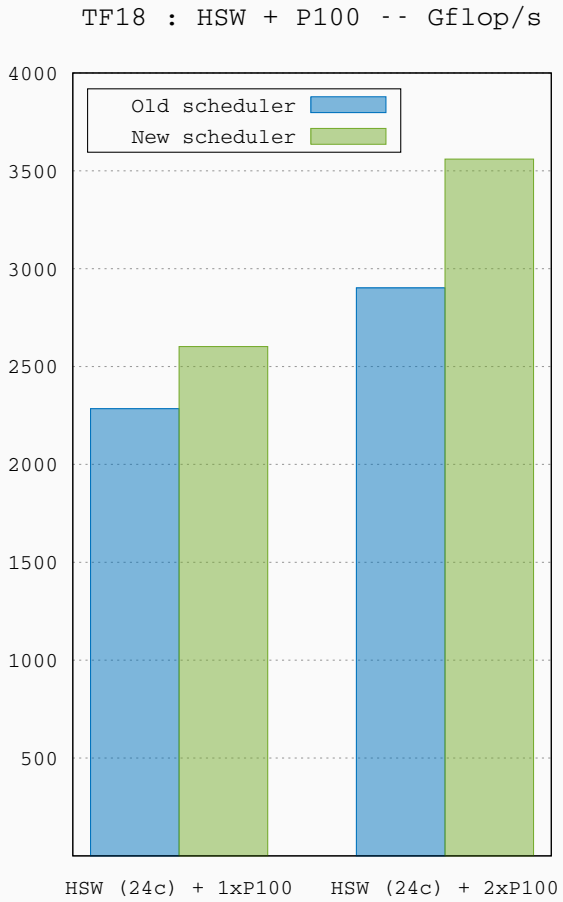
Improved scheduler (work in progress)



- Resource are grouped into families to improve data locality
- Work-stealing may happen between families
- A single worker queue feeds all the GPU streams
- Tasks are assigned to families at submission time according to a mapping

Improved scheduler (work in progress)





- Resource are grouped into families to improve data locality
- Work-stealing may happen between families
- A single worker queue feeds all the GPU streams
- Tasks are assigned to families at submission time according to a mapping

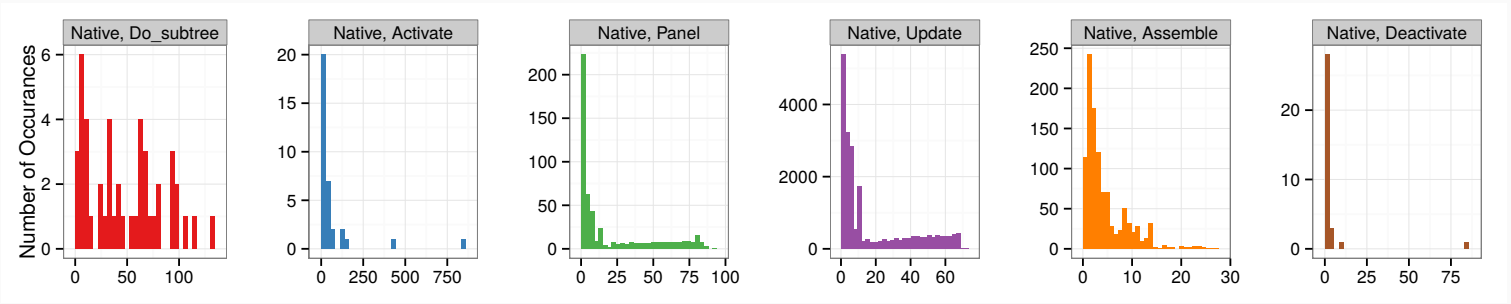
Simulation of qr_mumps on top of StarPU-SimGrid: 1D code

Simulating sparse solvers

Porting `qr_mumps` on top of SimGrid[1]

- Changing `main` for the subroutine
- Changing compilation process
- Careful kernel modeling as matrix dimension keeps changing

- Dense kernels during a single experiment are always executed with the same block/tile size $\rightsquigarrow$ duration very stable
- Sparse kernels depend on their input parameters $\rightsquigarrow$ more variability
- Cannot model sparse kernels with simple mean values



[1] s.a.b.g.ea:15.

Example for modeling kernels: `Panel`

- Theoretical `Panel` complexity:

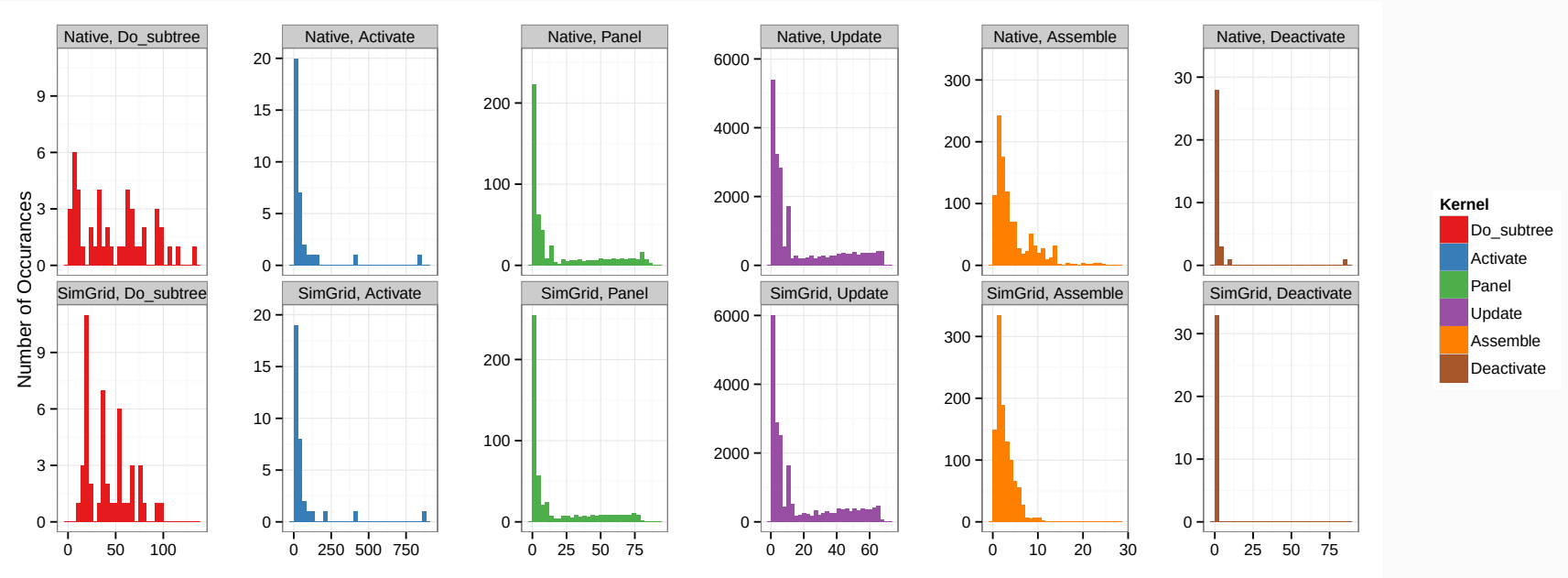
$$T_{\texttt{Panel}} = a + 2b(NB^2 \times MB) - 2c(NB^3 \times BK) + \frac{4d}{3}NB^3$$

- We can do a linear regression based on ad hoc calibration

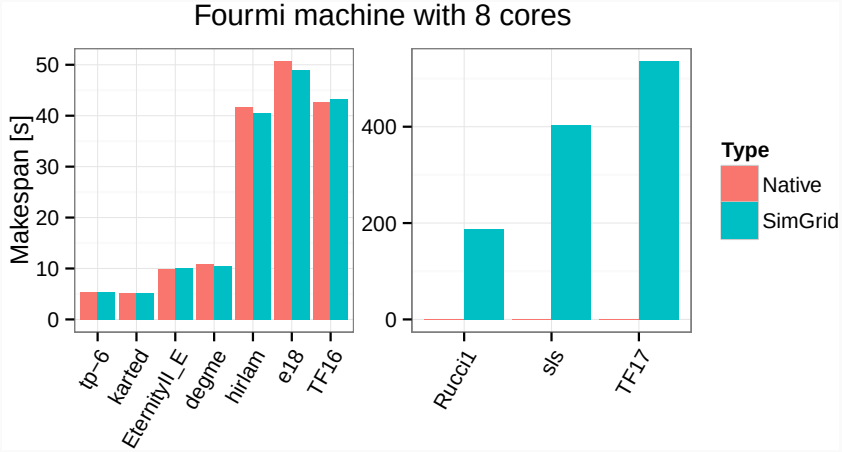
	Panel Duration	
Constant	-2.49×10^{1} $(-2.83 \times 10^{1}, -2.14 \times 10^{1})$	***
$NB^2 \times MB$	5.49×10^{-7} $(5.46 \times 10^{-7}, 5.51 \times 10^{-7})$	***
$NB^3 \times BK$	-5.52×10^{-7} $(-5.57 \times 10^{-7}, -5.48 \times 10^{-7})$	***
NB^3	1.50×10^{-5} $(1.30 \times 10^{-5}, 1.70 \times 10^{-5})$	***
Observations	493	
R^2	0.999	
Note:	*p<0.1; **p<0.05; ***p<0.01	

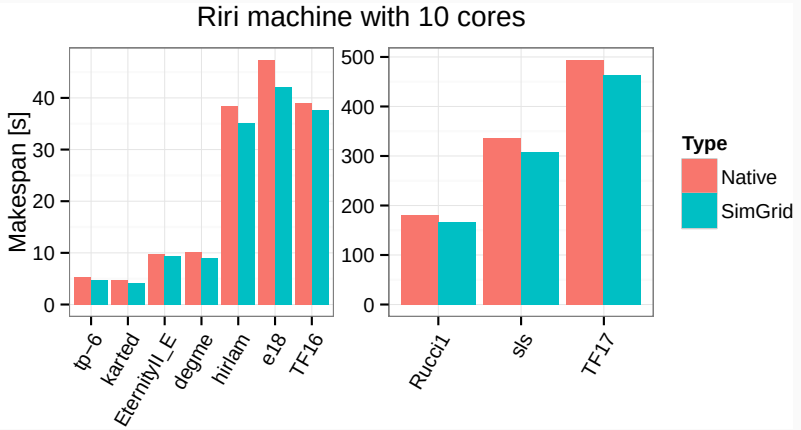
Comparing kernel duration distributions

	Do_subtree	activate	Panel	Update	Assemble
1.	#Flops	#Zeros	*NB*	*NB*	#Coeff
2.	#Nodes	#Assemble	*MB*	*MB*	/
3.	/	/	*BK*	*BK*	/
R^2	0.99	0.99	0.99	0.99	0.86

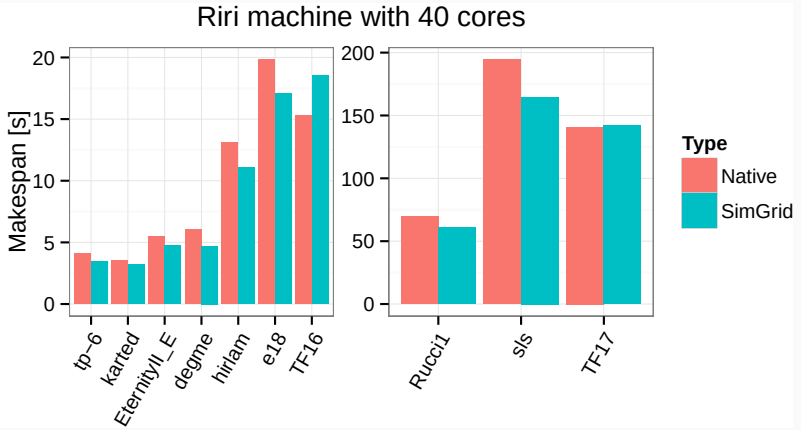


Overview of simulation accuracy



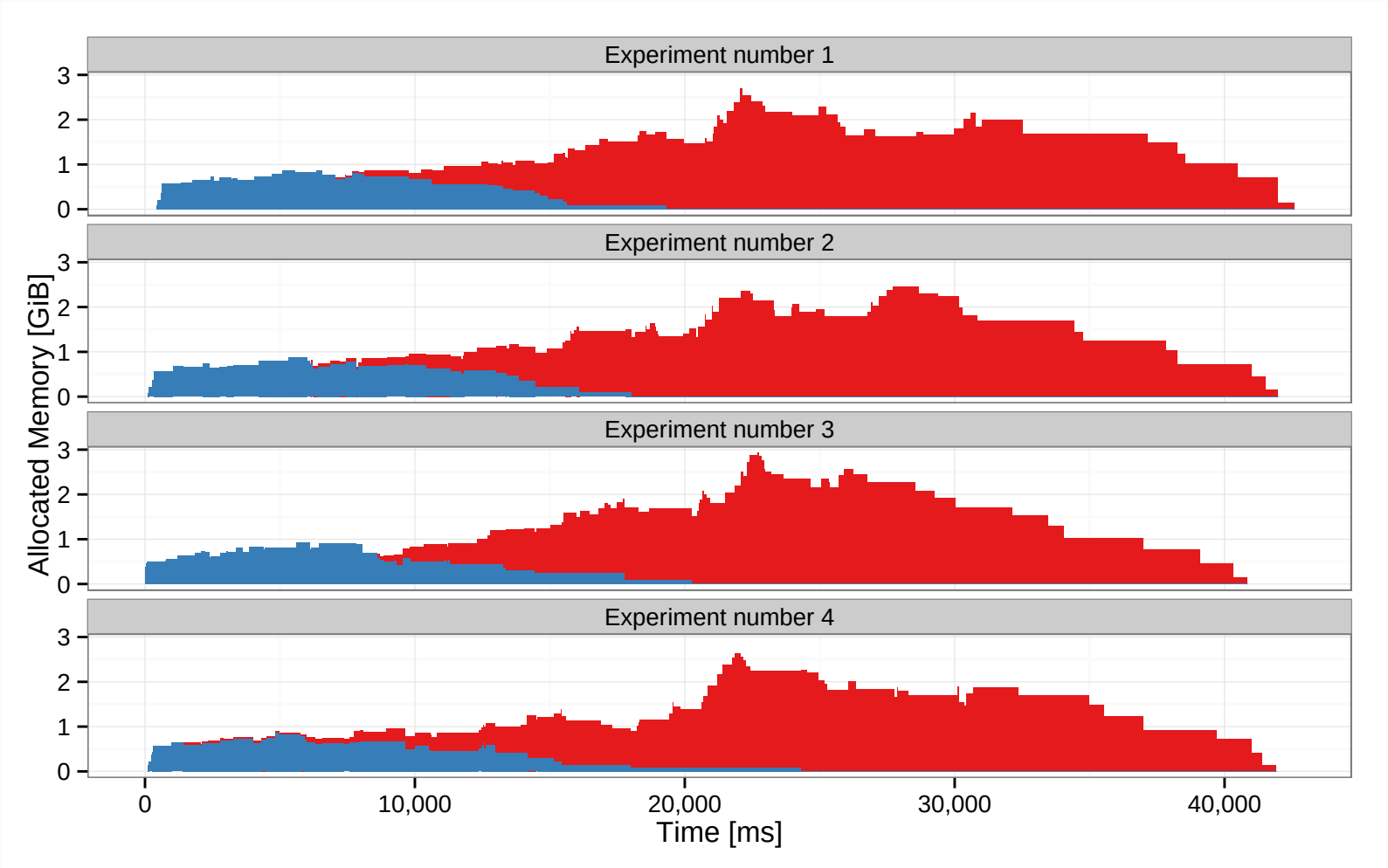


- Most of the time, simulation is slightly optimistic
- With bigger and architecturally more complex machines, error increases



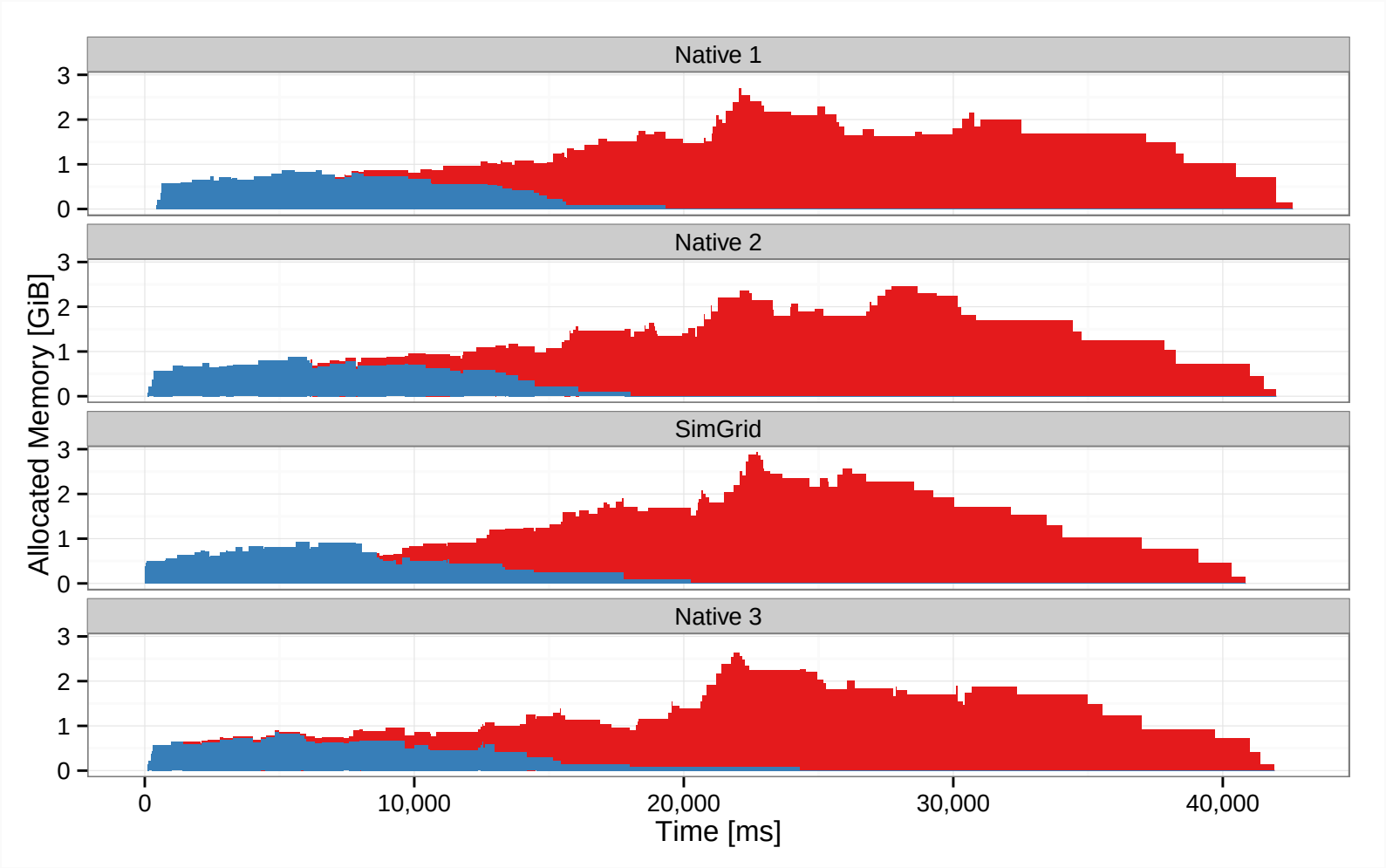
Studying memory consumption

- Minimizing memory footprint is often critical
- Remember scheduling is dynamic so consecutive Native experiments have different output



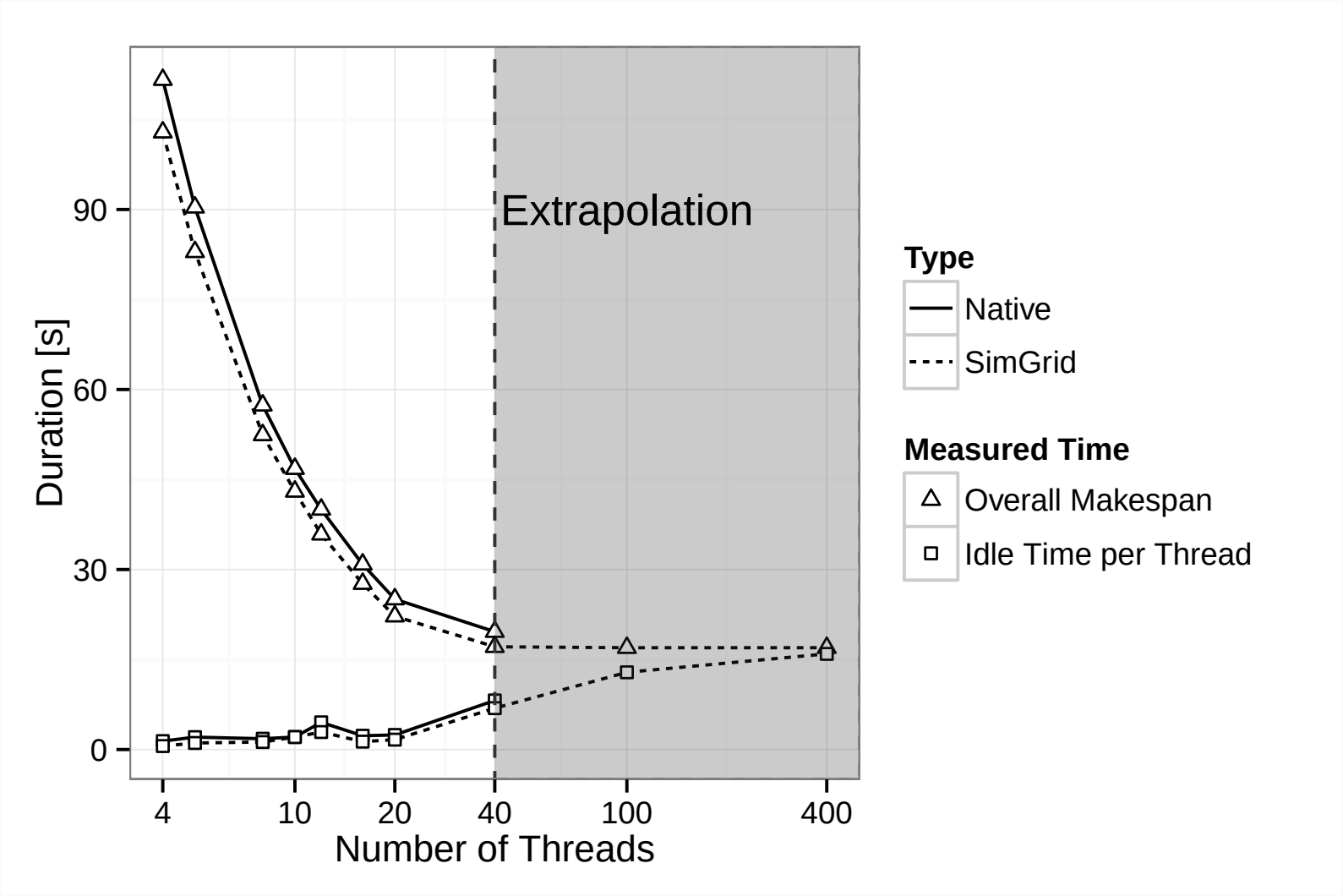
Studying memory consumption

- Minimizing memory footprint is often critical
- Remember scheduling is dynamic so consecutive Native experiments have different output



Extrapolating to larger machines

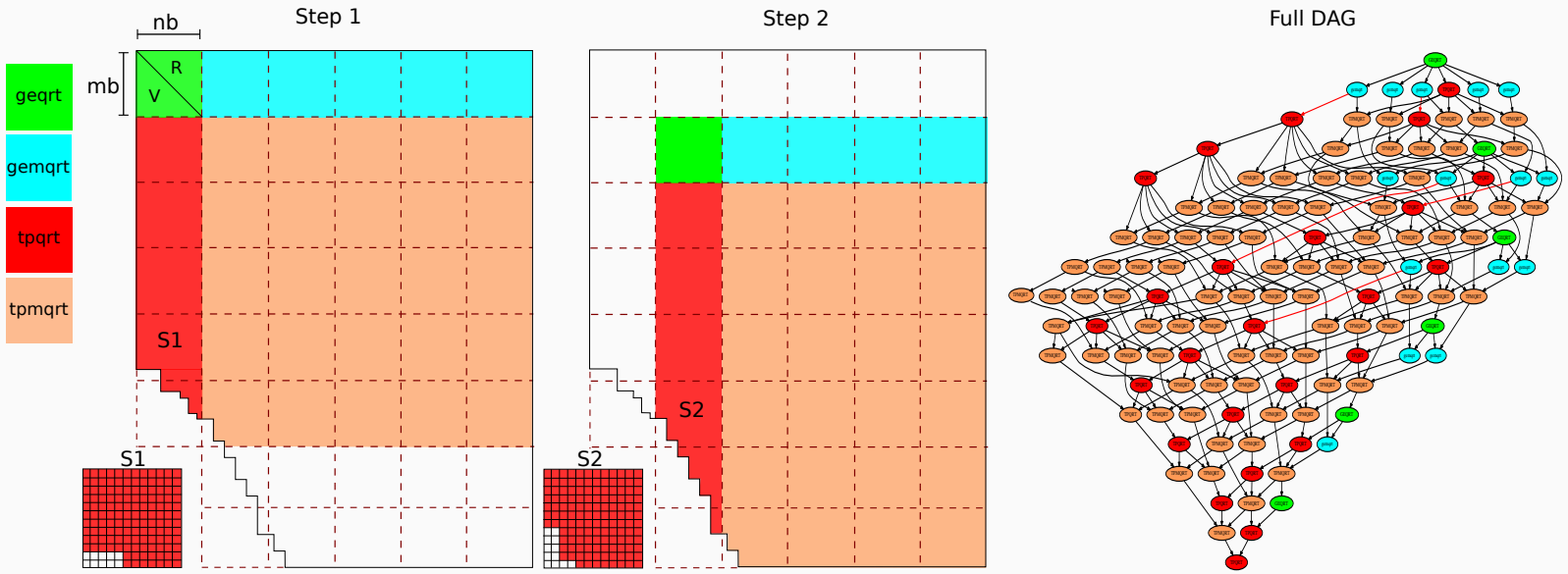
- Predicting performance in idealized context
- Studying the parallelization limits of the problem



simulation: 2d fully-featured code (work in progress)

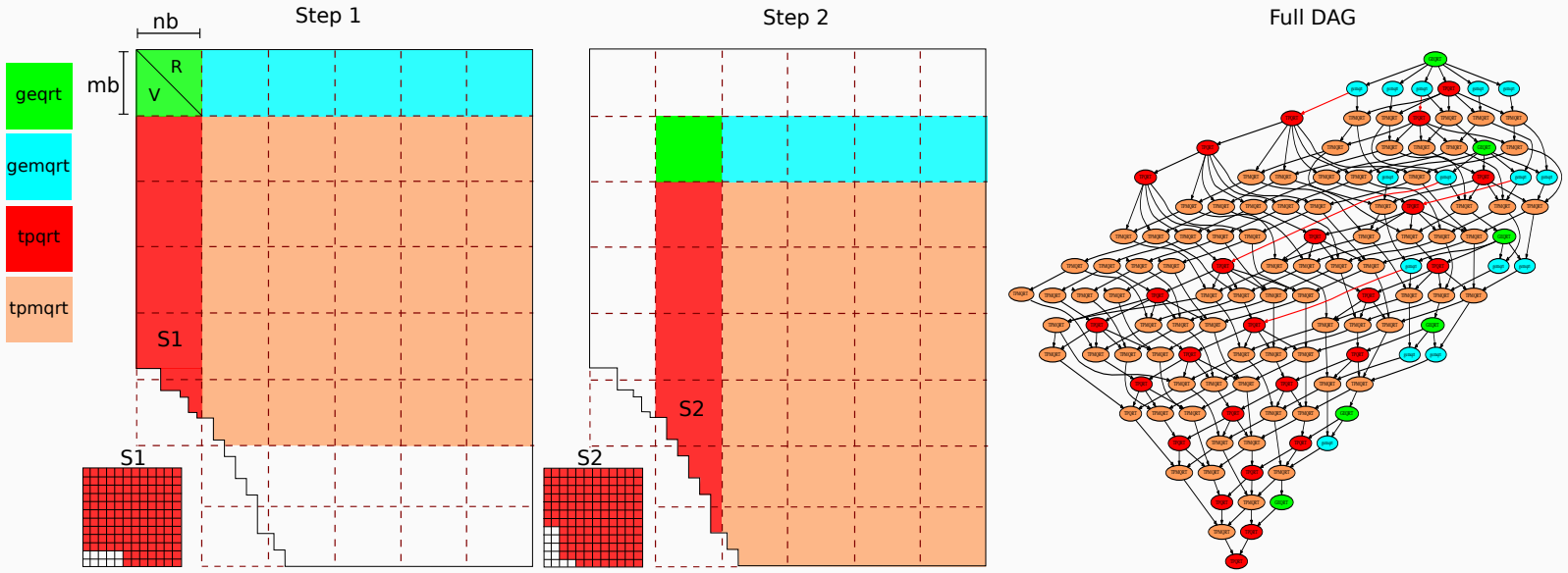
2D fully-featured code

- Increase concurrency through 2D ("tile") algorithms
- Exploit sparsity of the staircase structure within fronts



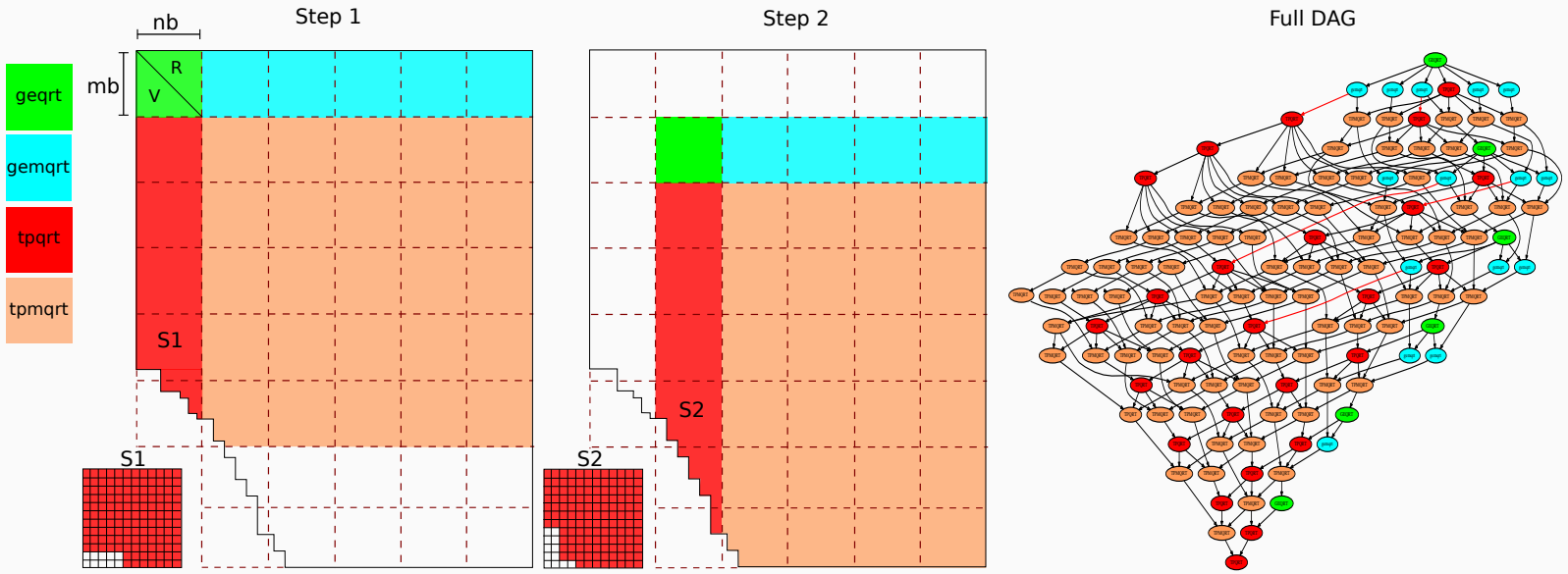
2D fully-featured code

- Increase concurrency through 2D ("tile") algorithms
- Exploit sparsity of the staircase structure within fronts
- Focus on the update (tpmqrt) kernel



2D fully-featured code

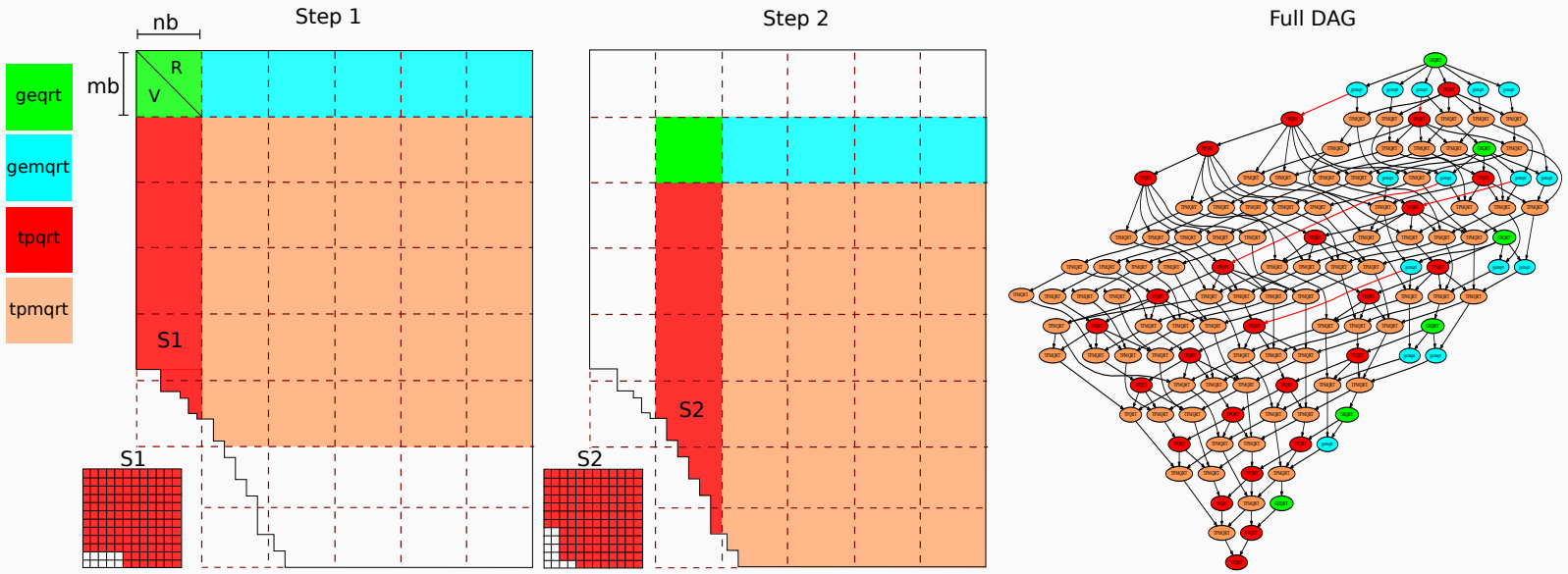
- Increase concurrency through 2D ("tile") algorithms
- Exploit sparsity of the staircase structure within fronts



How to handle the staircase structure?

2D fully-featured code

- Increase concurrency through 2D ("tile") algorithms
- Exploit sparsity of the staircase structure within fronts

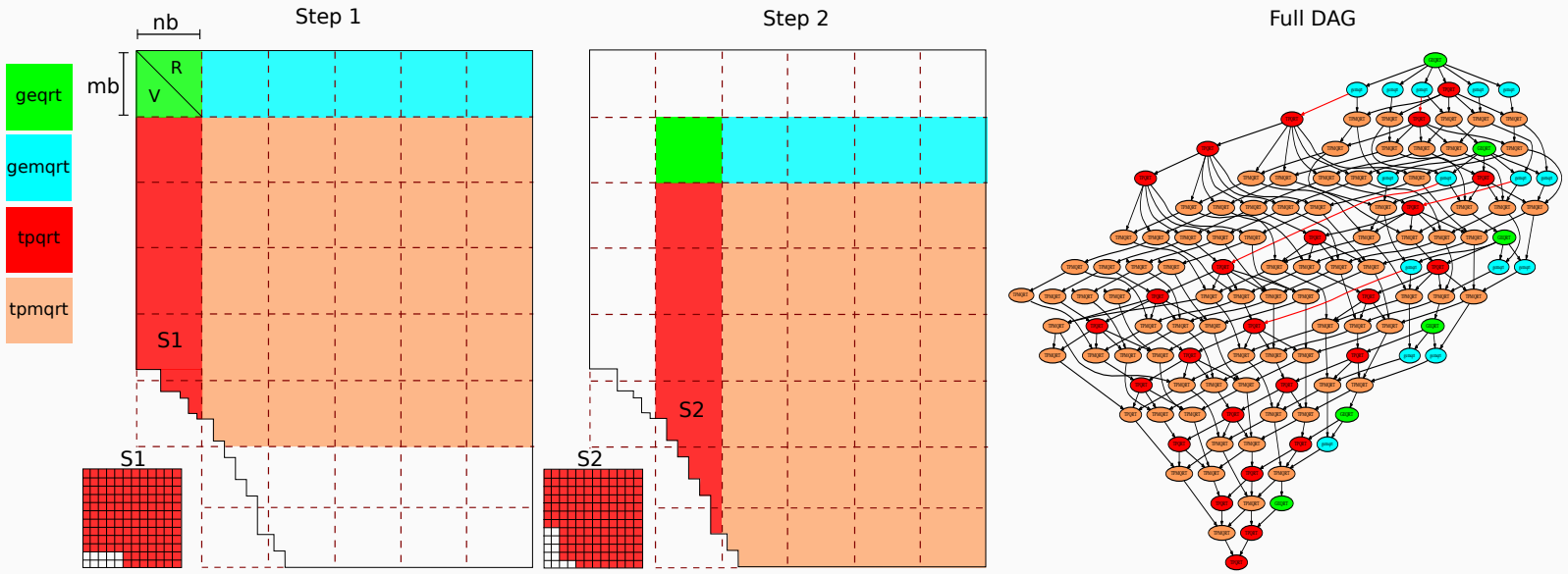


How to handle the staircase structure?

- S1={10,10,10,10,10,12,12,12,12,12,12,12}

2D fully-featured code

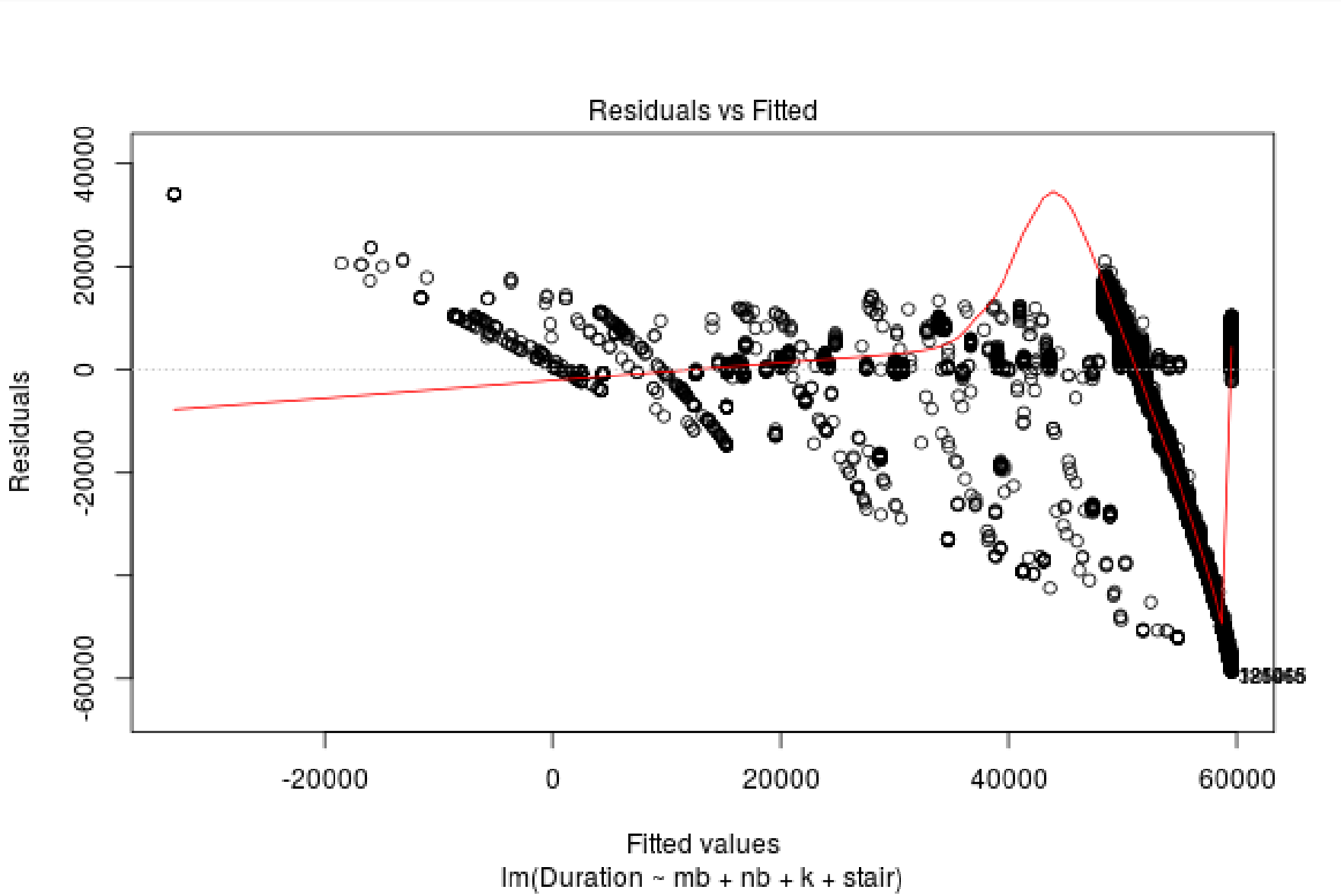
- Increase concurrency through 2D ("tile") algorithms
- Exploit sparsity of the staircase structure within fronts



How to handle the staircase structure?

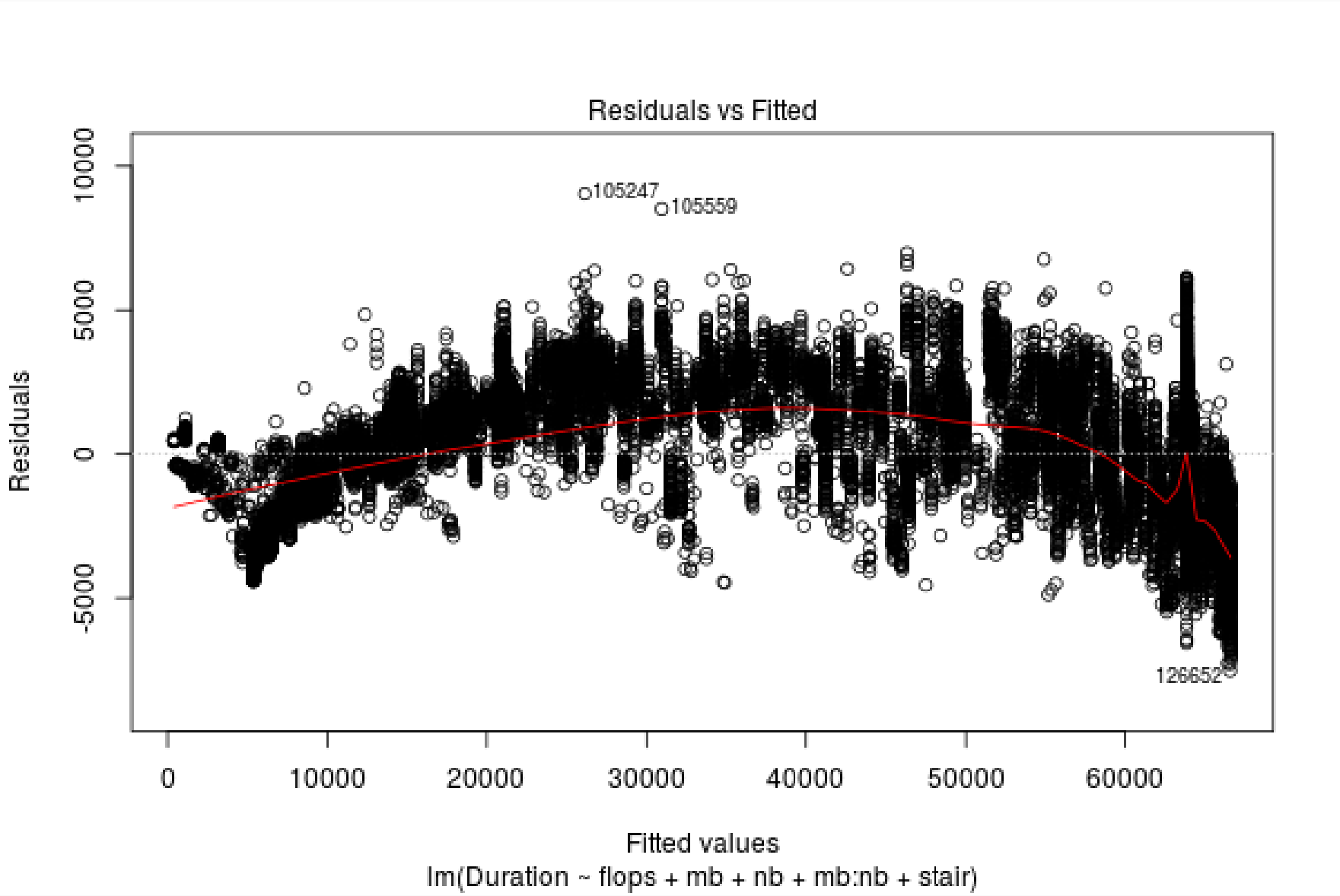
- S1={10,10,10,10,10,12,12,12,12,12,12,12}
- stair $= 10 * 5 + 12 * 7 = 134$

Model m1: $lm(Duration \sim stair + mb + nb + k)$



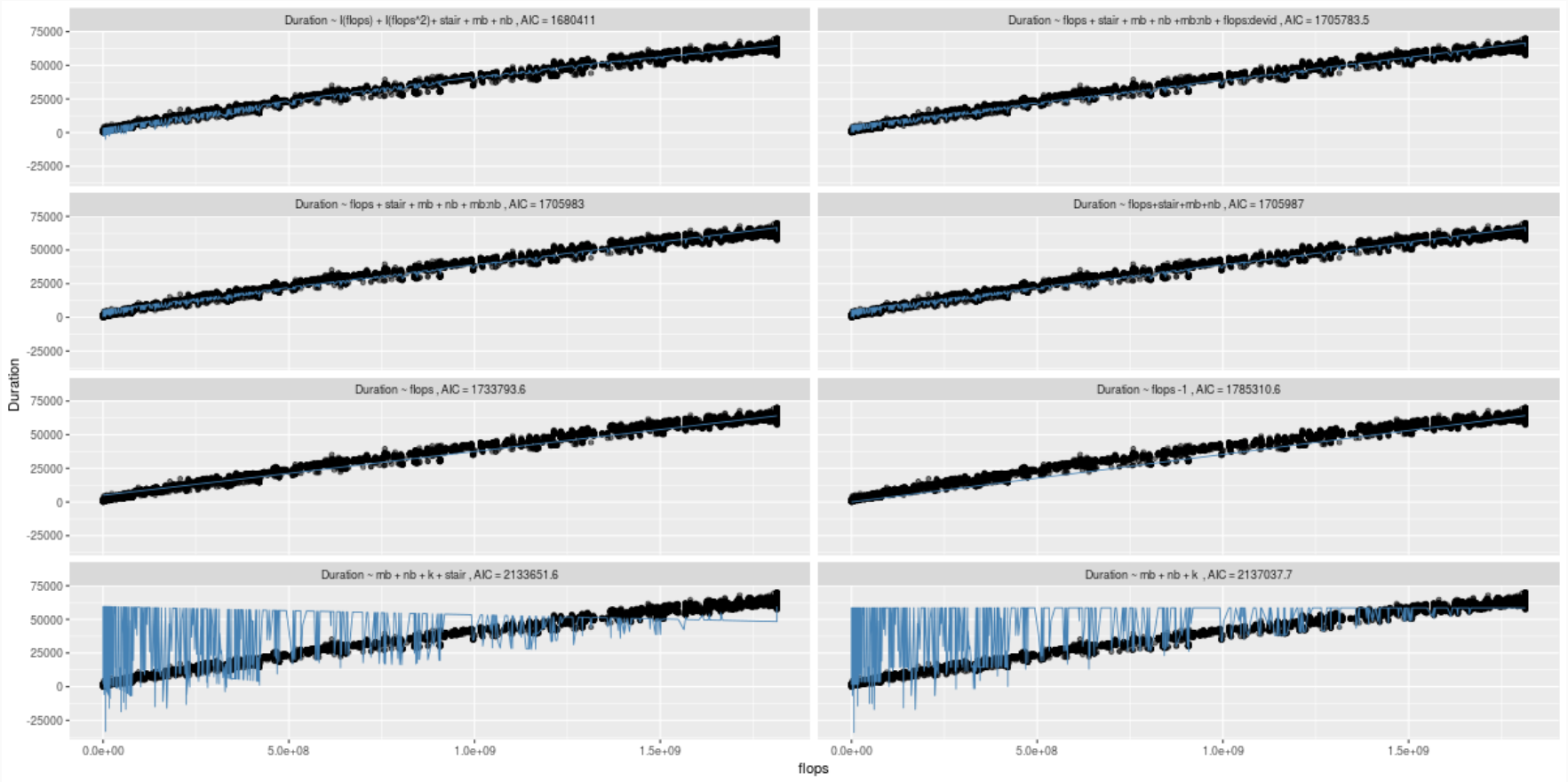
Adjusted R^2: 0.26

Model m2: $lm(Duration \sim flop + stair + mb + nb + mb : nb)$

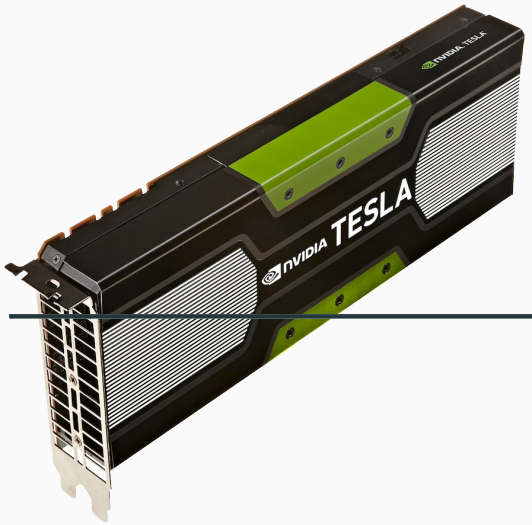


Adjusted R^2: 0.997

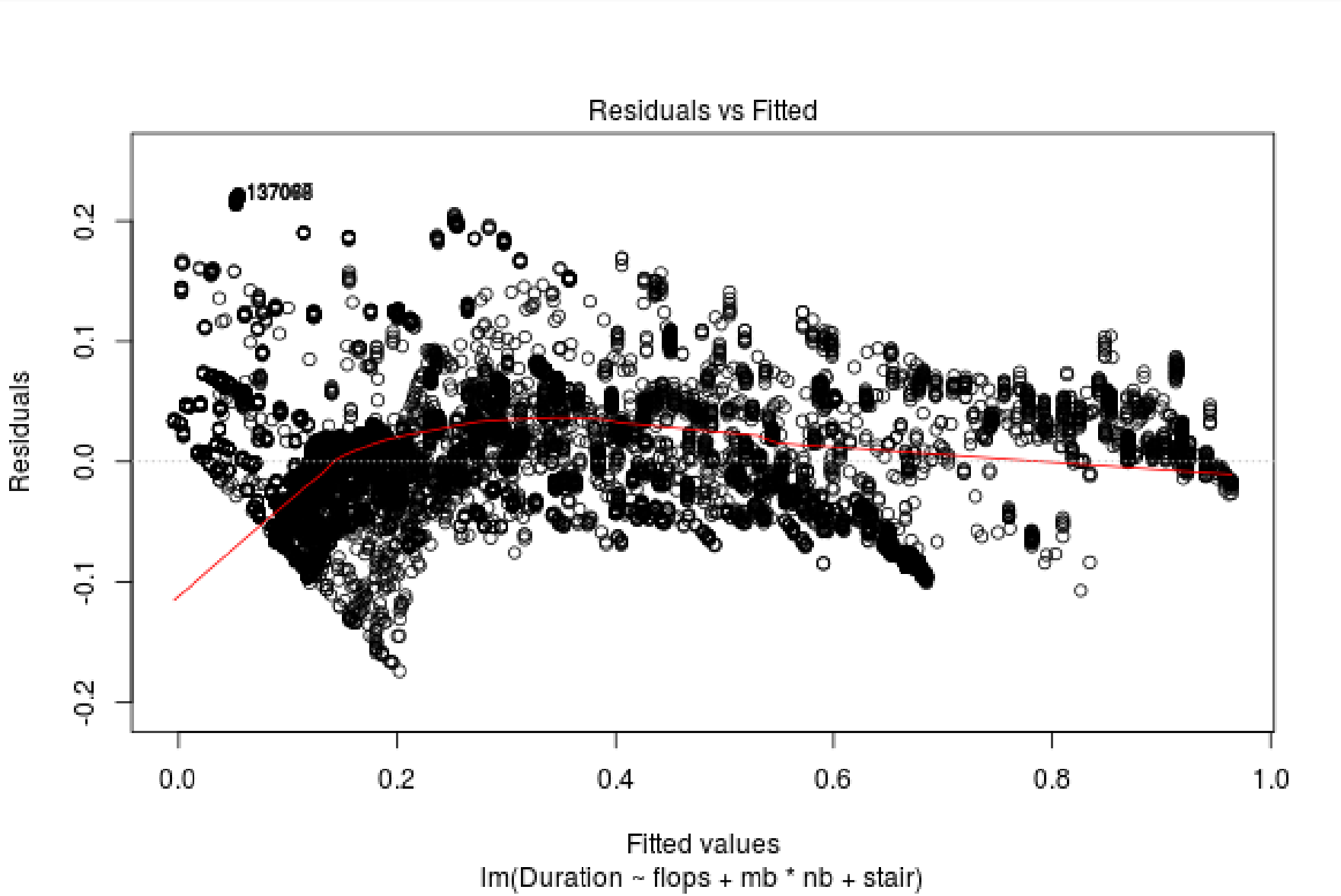
Models for tpmqrt kernel ordered wrt AIC



SIMULATION: GPU-BASED SYSTEMS (IN PROGRESS)

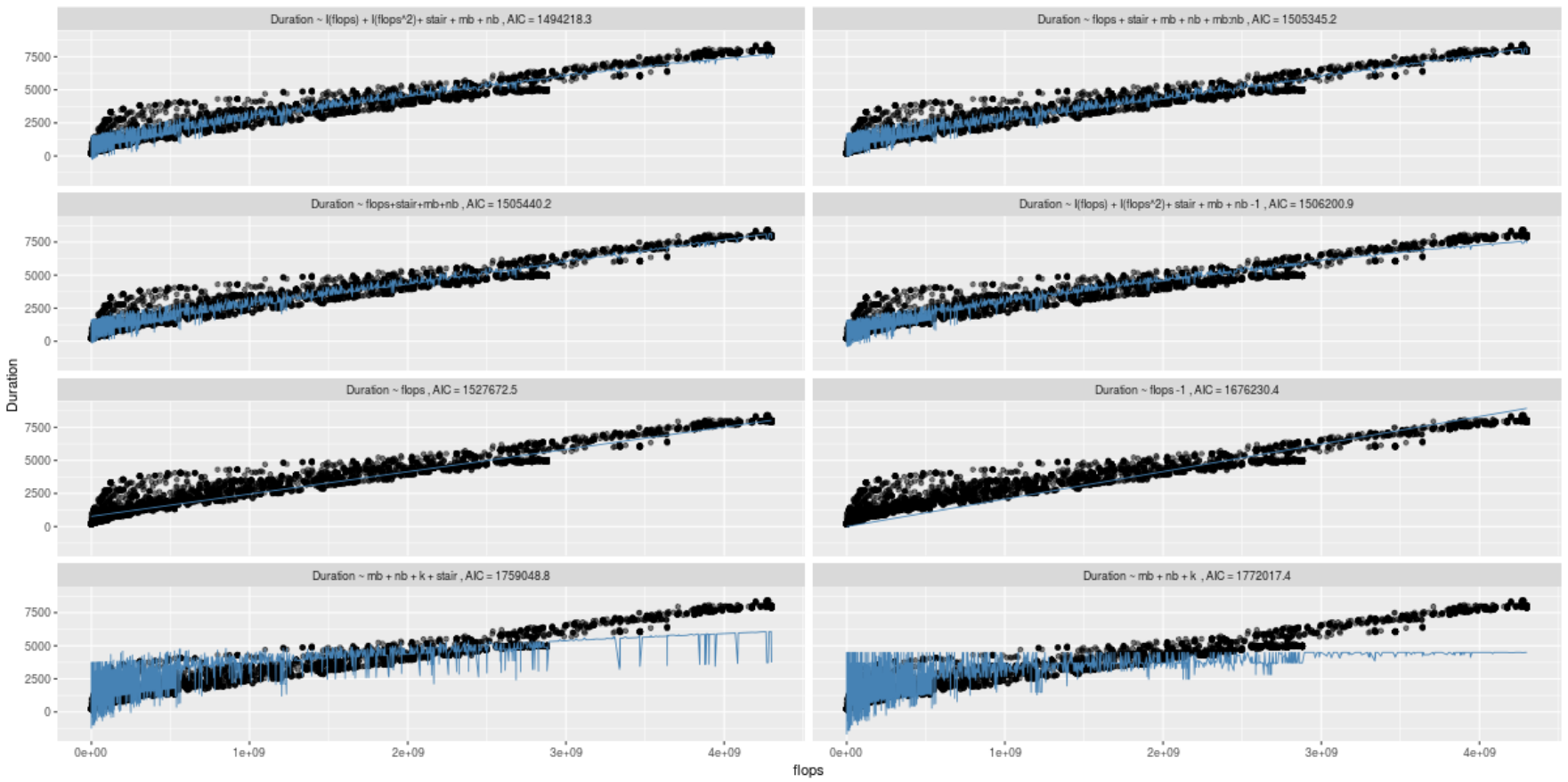


Model m2: $lm(Duration \sim flop + stair + mb + nb + mb : nb)$

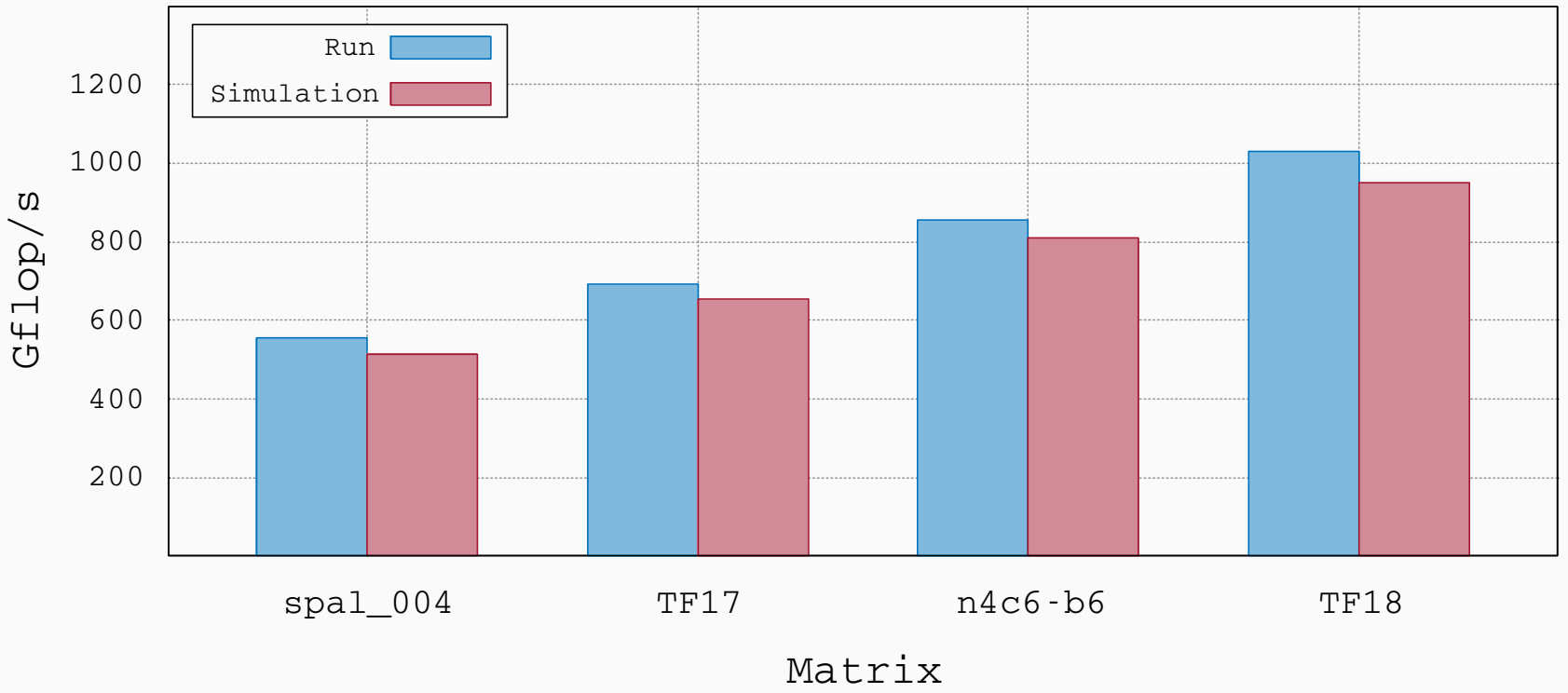


Adjusted R^2: 0.92

Models for tpmqrt kernel on GPU K40 ordered wrt AIC

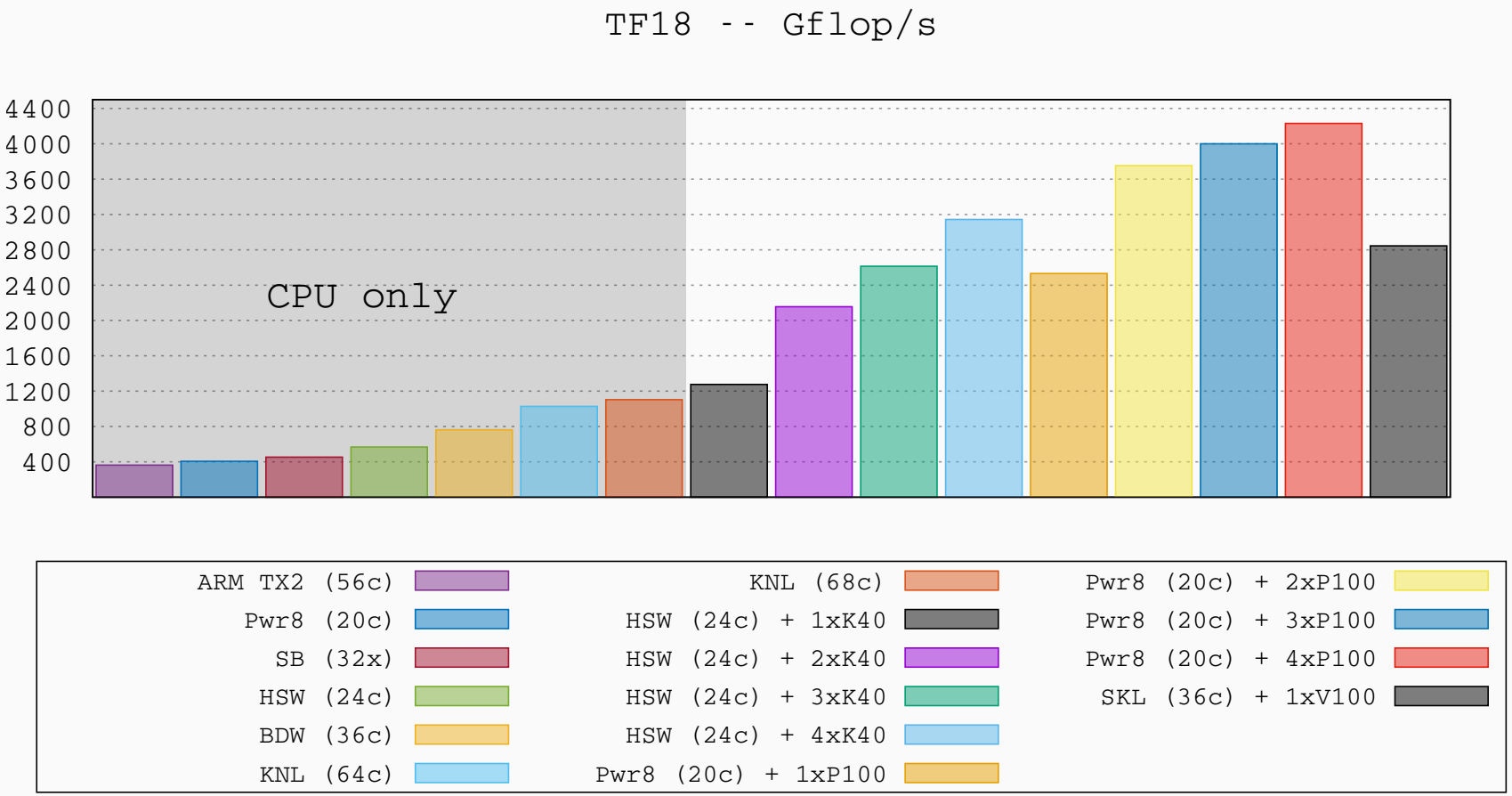


Overall simulation of an 24 cores + 1 GPU



CONCLUDING REMARKS

Experimental results



Thanks to IDRIS, PlaFRIM, CALMIP and GENCI for providing access to the resources

References I

Thanks!
Questions?

Other features

Memory awareness

The memory footprint of the multifrontal method increases when executed in parallel. We have developed a deadlock free memory capping technique that allows for achieving the parallel factorization within a prescribed memory envelope

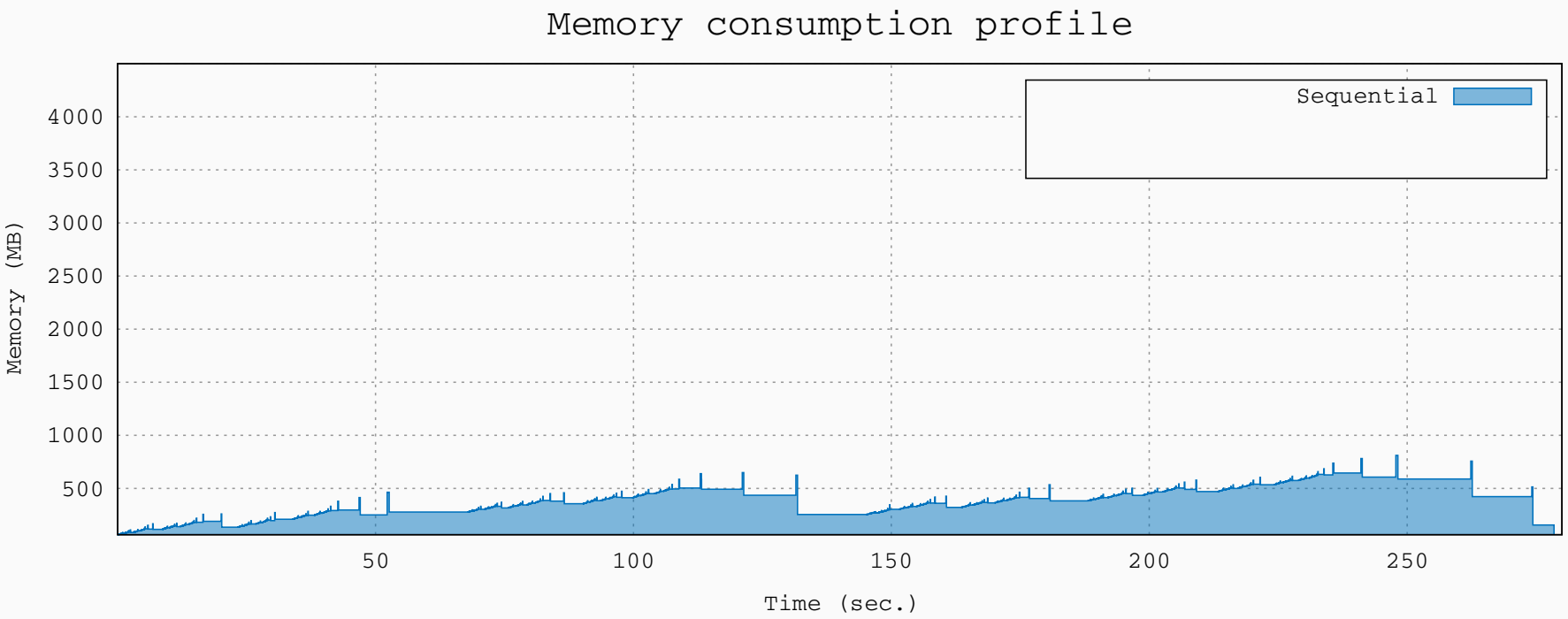
Performance analysis

Because StarPU has full control of the workload, it can produce accurate performance measures. Based on these we have developed a method for detailed performance analysis

Matrices from the Suite Sparse Matrix Collection

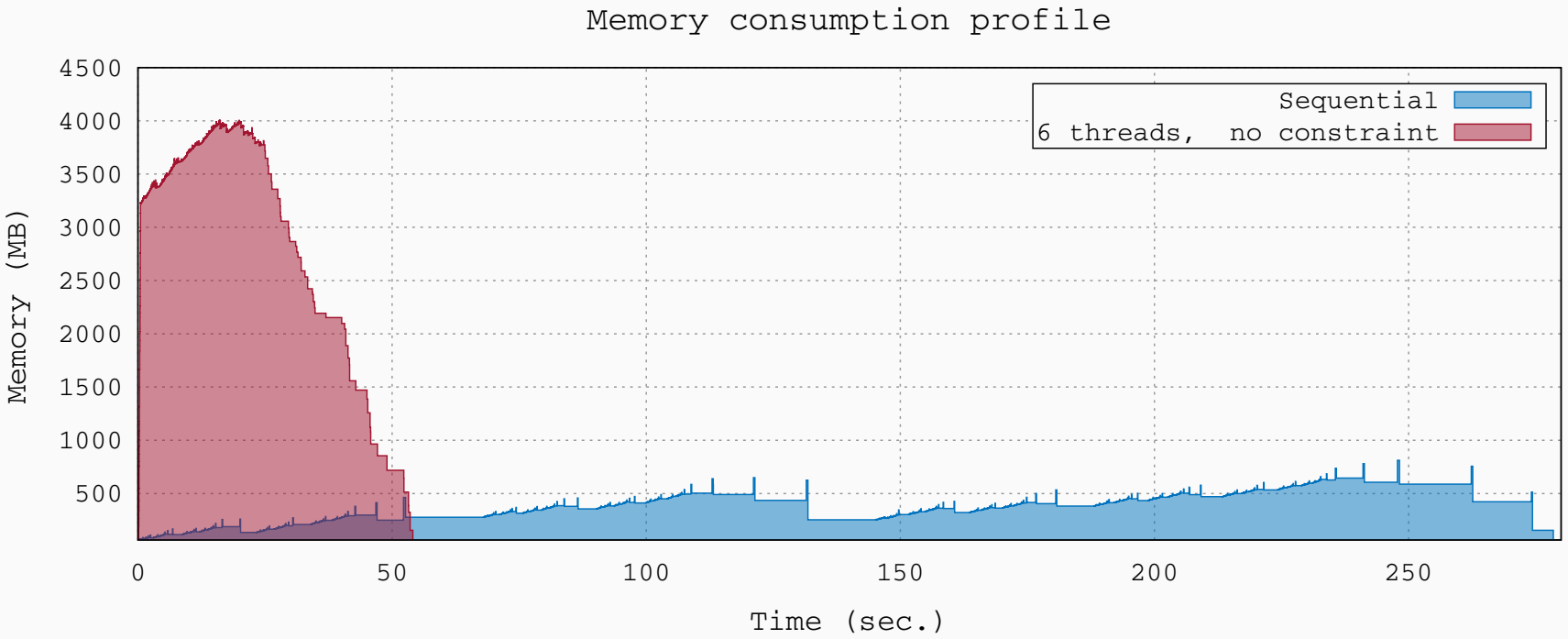
#	Mat. name	m	n	nz	op. count
12	hirlam	1385K	452K	2713K	1384G
13	flower_8_4	55K	125K	375K	2851G
14	Rucci1	1977K	109K	7791K	5671G
15	ch8-8-b3	117K	18K	470K	10709G
16	GL7d24	21K	105K	593K	16467G
17	neos2	132K	134K	685K	20170G
18	spal_004	10K	321K	46168K	30335G
19	n4c6-b6	104K	51K	728K	62245G
20	sls	1748K	62K	6804K	65607G
21	TF18	95K	123K	1597K	194472G
22	lp_nug30	95K	123K	1597K	221644G
23	mk13-b5	135K	270K	810K	259751G

Memory footprint in the multifrontal method



- In sequential: the memory consumption varies greatly because fronts are allocated and deallocated dynamically. The maximum memory is referred to as the sequential peak M_s.

Memory footprint in the multifrontal method



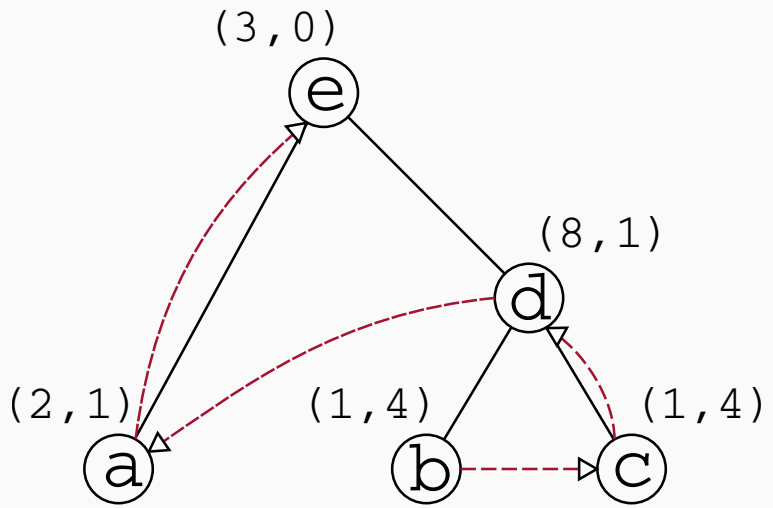
- In sequential: the memory consumption varies greatly because fronts are allocated and deallocated dynamically. The maximum memory is referred to as the sequential peak M_s.
- In parallel: the peak memory consumption M_p can be much higher because of tree parallelism.

Task scheduling under memory constraint

Memory-aware parallel execution

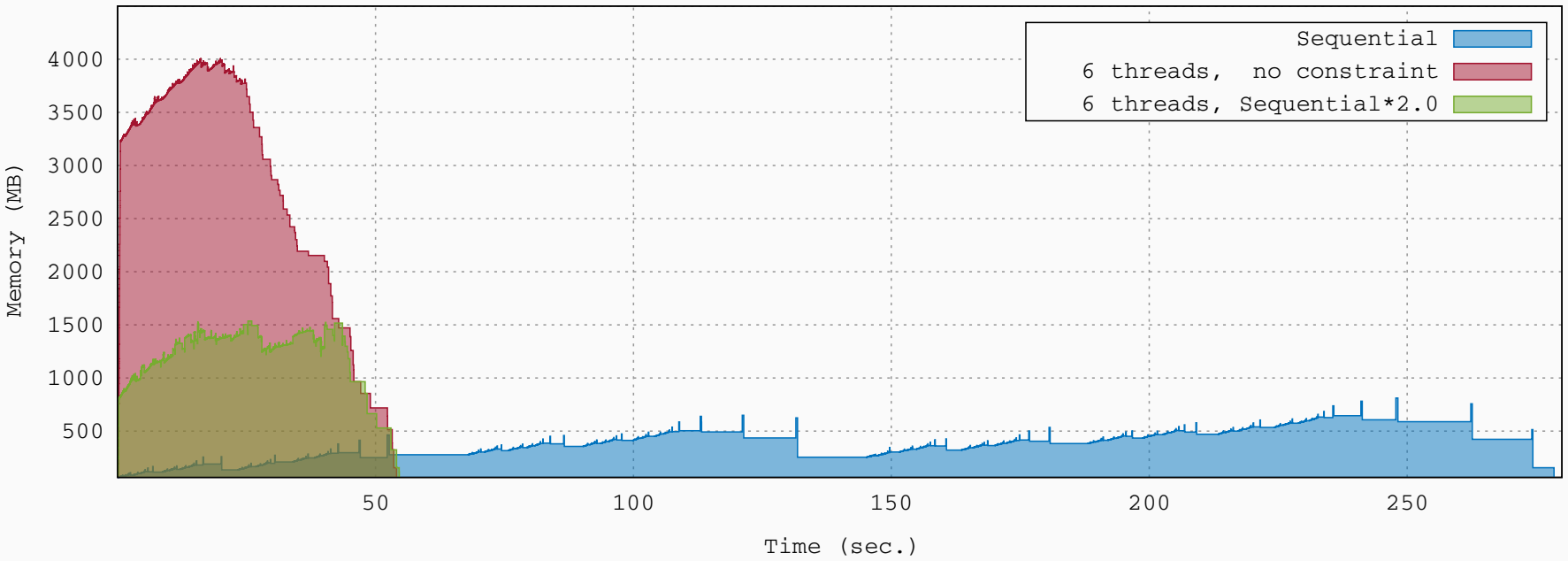
Objective: achieve efficient parallel execution within a prescribed memory consumption $M_p \leq \alpha M_s, \quad \alpha \geq 1$. Method: suspend tasks submission when no more memory is available and resume it when enough memory has been freed by previously submitted tasks.

Memory deadlock prevention by ensuring fronts are allocated in the same order as in sequential: straightforward to achieved thanks to the Sequential Task Flow model.



See also related work by Agullo *et al.*, Marchal *et al.* and Amestoy *et al.* on memory-aware scheduling and memory deadlock prevention.

Task scheduling under memory constraint



- Tighter memory bound $\rightarrow$ less concurrency $\rightarrow$ slower execution.
- In practice the execution time is increased only for very small matrices or very narrow/unbalanced elimination trees.