Towards a multifrontal QR factorization for heterogeneous architectures over runtime systems

Preliminary work on multicore architectures

Florent Lopez, Joint work with IRIT Toulouse, LaBRI / Inria Bordeaux, LIP / Inria Lyon

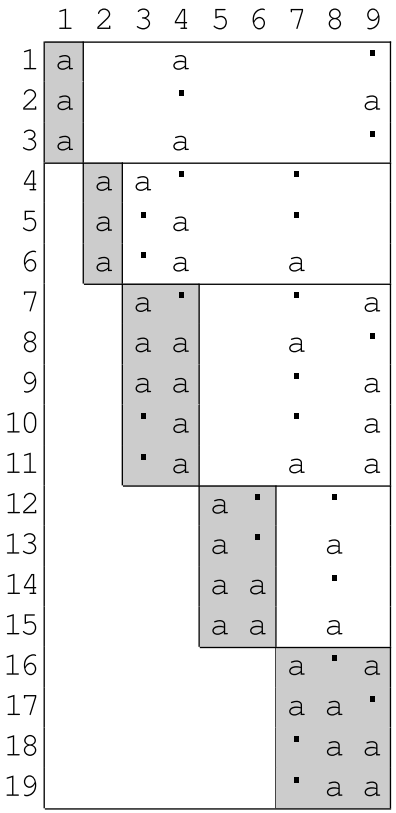
MUMPS Users Group Meeting, May 29-30, 2013

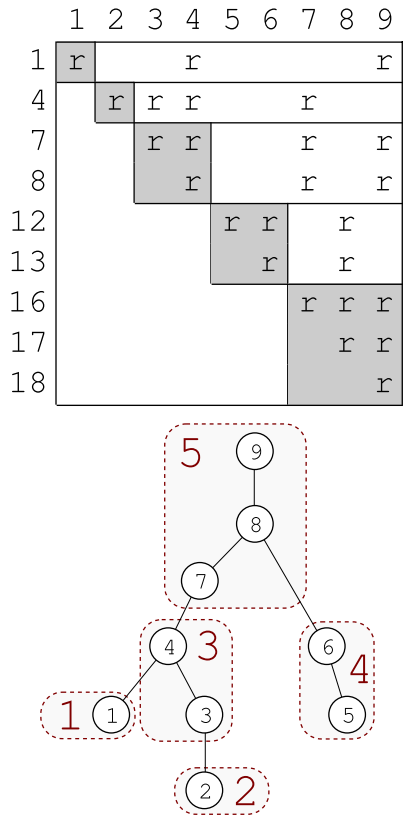
Context of the work

The Multifrontal method

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





Accelerated architectures

Accelerated architectures including GPUs are extremely popular in the HPC community:

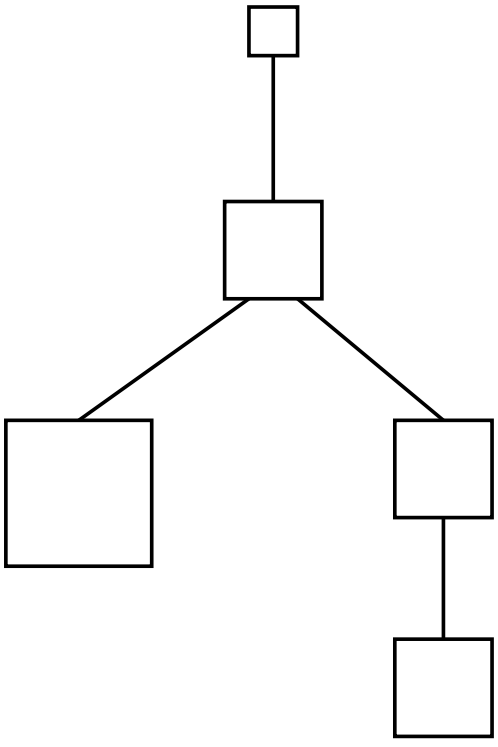
- high density of computational units clocked at low frequencies (wrt the latest generation of CPUs)
  - high potential performance peak for parallel applications
  - limited power consumption

New generation of accelerators: MIC (Many Integrated Core) architecture

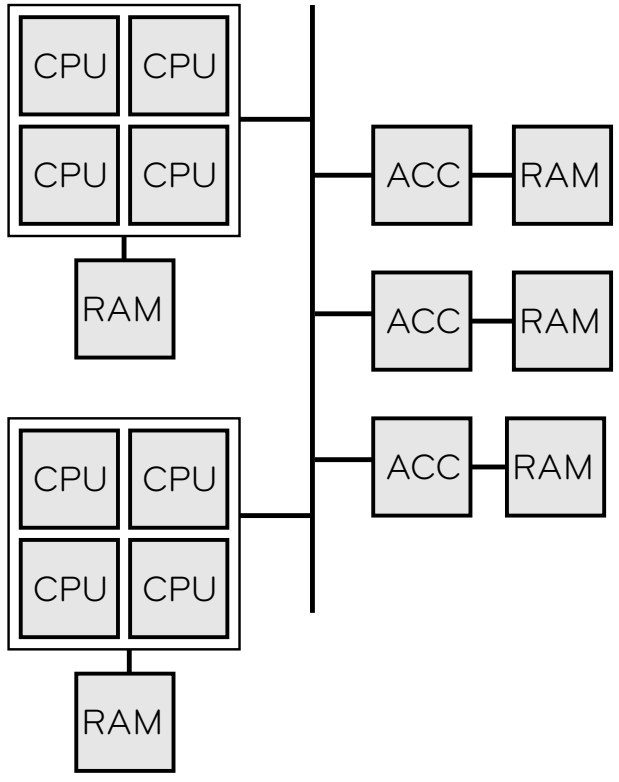
- 60 cores (In-order core derived from Pentium: energy efficient)
- clocked at 1053 MHz
- 240 threads (with 4 hyper threading sibling per core)
- advanced VPU per core (512-bit SIMD)

Incompatible programming models $\Rightarrow$ specific kernels and algorithms to achieve performance

Accelerated architectures (ACC: GPU or MIC)



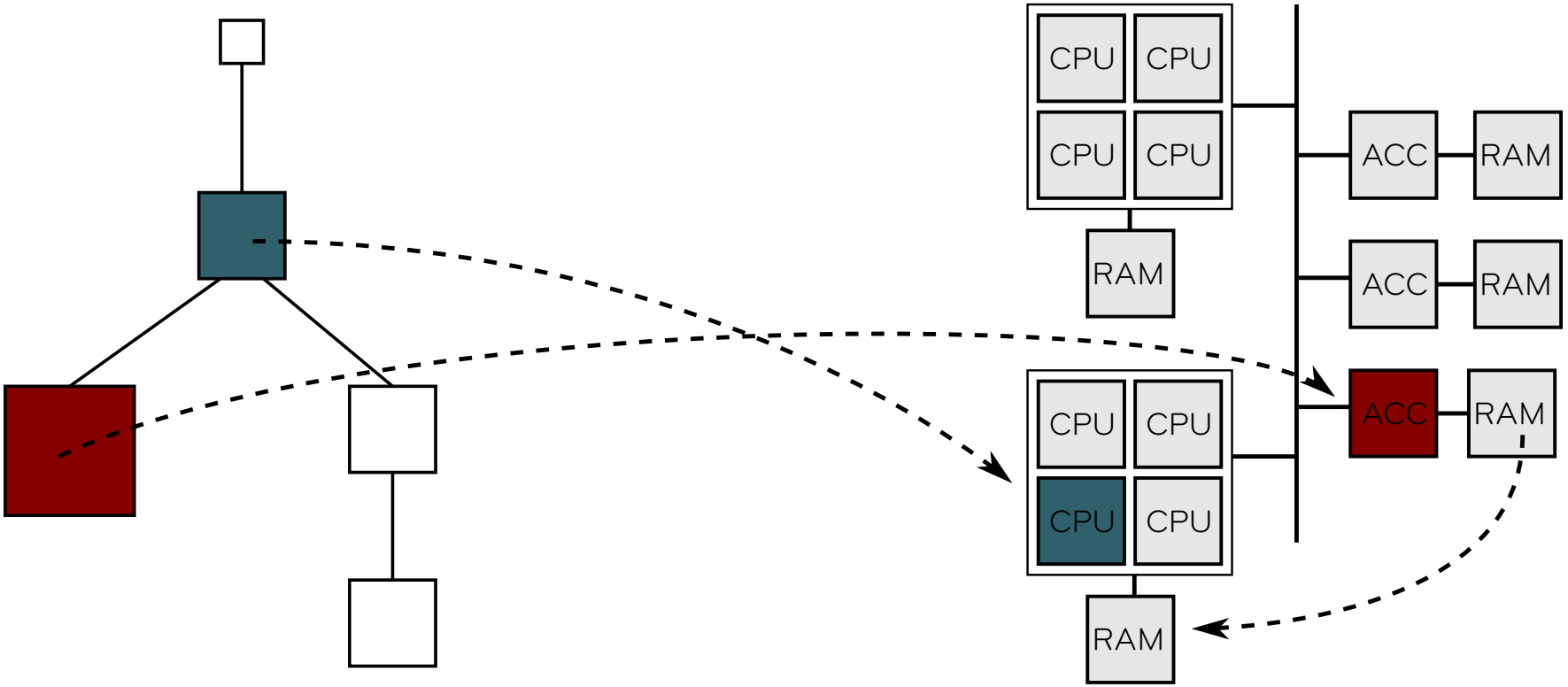
Elimination tree



Heterogeneous platform

- an extremely heterogeneous workload
- a heterogeneous architecture
- mapping tasks is challenging

Accelerated architectures (ACC: GPU or MIC)

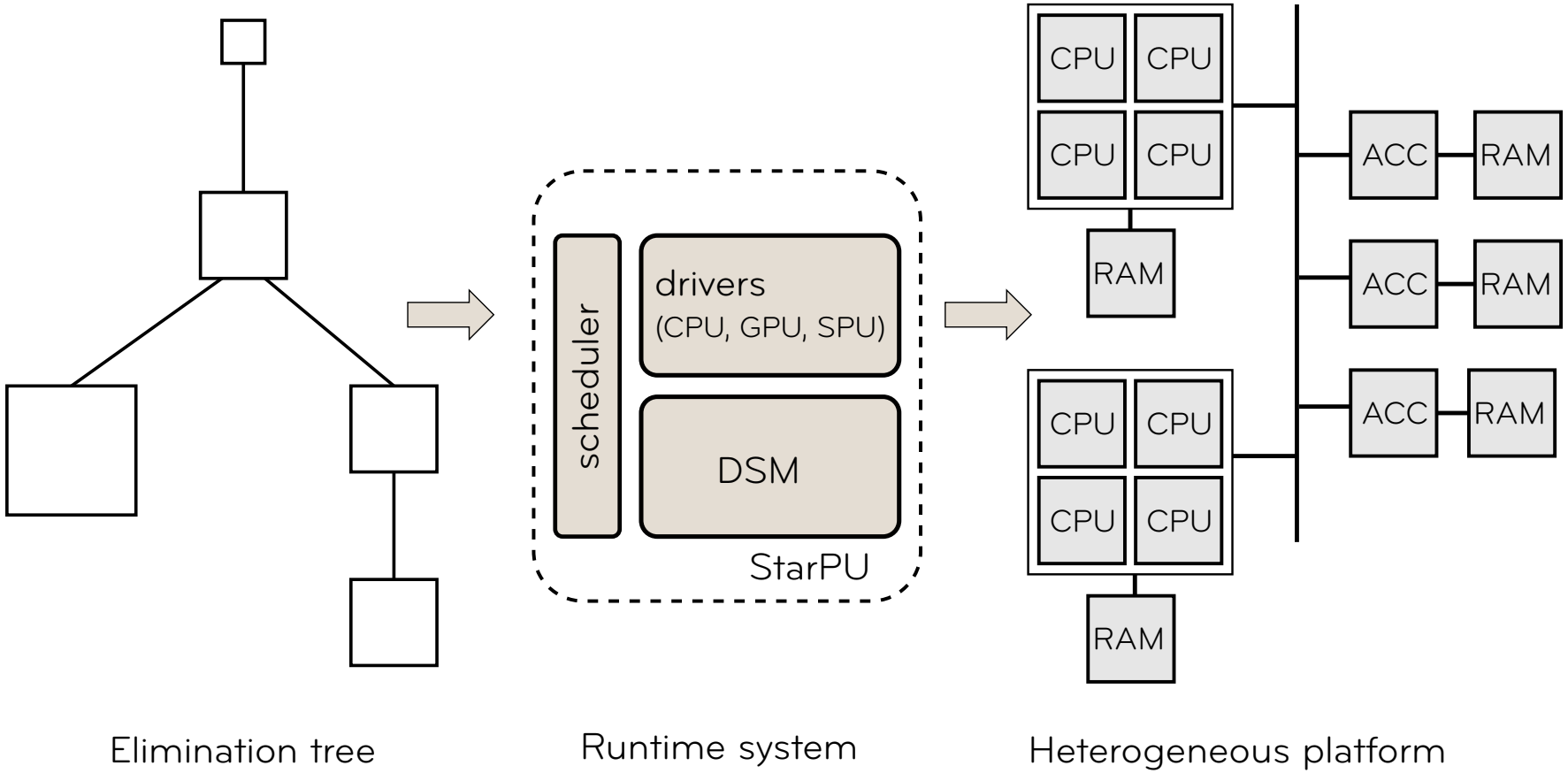


Elimination tree

Heterogeneous platform

- management of data consistency by hand
- architecture dependant approach
- difficult to maintain

Accelerated architectures (ACC: GPU or MIC)



Another option is to exploit the features of a modern runtime system capable of handling the scheduling and the data consistency in a dynamic way.

Runtime systems

Runtime system: abstract layer between application and machine with the following features:

- automatic detection of the task dependencies
- dynamic task scheduling on different types of processing units.
- management of *multi-versioned* tasks (an implementation for each type of processing unit)
- consistency management of manipulated data.

Runtime systems

Objective of the study: evaluate the usability and the effectiveness of a general purpose runtime system with complex and irregular workload such as a sparse factorization

- exploitation of GPU-accelerated architectures

This approach is widely adopted in the case of dense linear algebra:

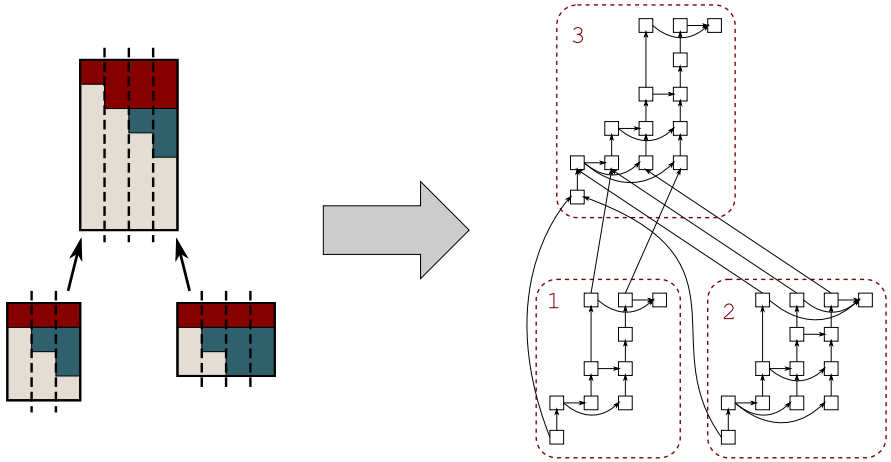
- PLASMA (QUARK)
- DPLASMA (PaRSEC)
- MAGMA-MORSE (StarPU)
- FLAME (SuperMatrix)

it is challenging for complex and irregular problems such as sparse linear algebra (related work on PaStiX with Pierre Ramet)

Multifrontal QR factorization on multicores over StarPU

parallelism and scheduling strategy in qr_mumps

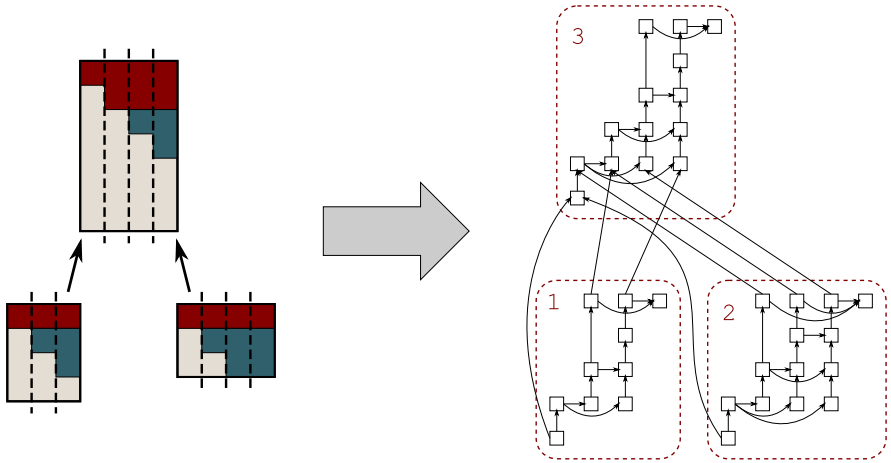
In qr_mumps node and tree parallelism are exploited consistently, by partitioning the frontal matrices and replacing the elimination tree with a DAG:



the scheduler efficiency is constrained by the tasks search-space: the scheduling complexity depends on the number of active fronts and therefore is not very scalable.

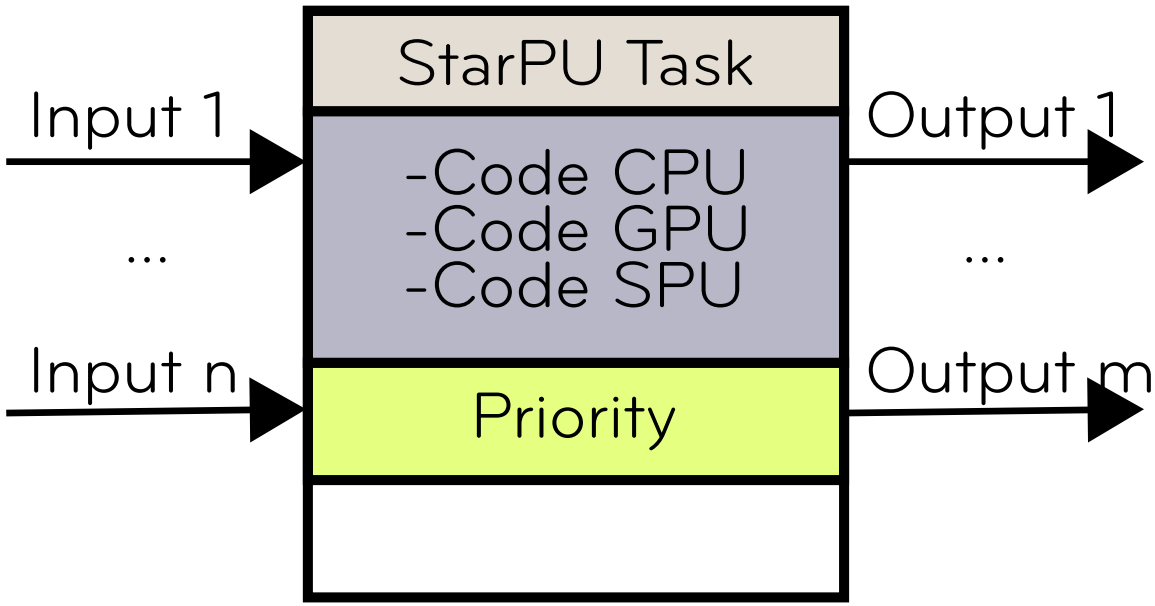
parallelism and scheduling strategy in qr_mumps

In qr_mumps node and tree parallelism are exploited consistently, by partitioning the frontal matrices and replacing the elimination tree with a DAG:



the scheduler efficiency is constrained by the tasks search-space: the scheduling complexity depends on the number of active fronts and therefore is not very scalable.
Replace the ad hoc scheduler in qr_mumps with a general purpose runtime system

The multifrontal QR factorization: StarPU integration



- Depending on the input/output, StarPU detects the dependencies among tasks
- Depending on the availability of resources and the data placement, StarPU decides where to run a task

The multifrontal QR factorization: StarPU integration

Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: inout, B: in, $id = id_1$)
2: submit_task(fun_2, A: inout, $id = id_2$)
3: declare_dependency($id_3 \leftarrow id_1$)
4: submit_task(fun_1, C: inout, D: in,$id = id_3$)

The multifrontal QR factorization: StarPU integration

Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: inout, B: in, $id = id_1$)

detected dependency (data hazard)

explicit dependency

The multifrontal QR factorization: StarPU integration

Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: inout, B: in, $id = id_1$)
2: submit_task(fun_2, A: inout, $id = id_2$)

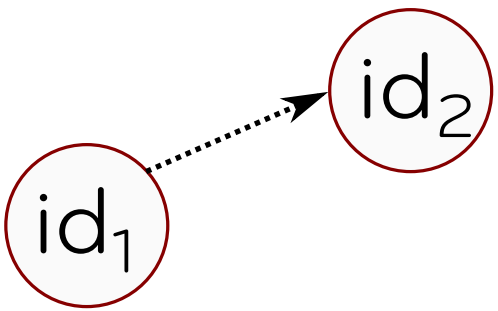
The multifrontal QR factorization: StarPU integration

Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: in**out**, B: in, $id = id_1$)
2: submit_task(fun_2, A: **in**out, $id = id_2$)



detected dependency (data hazard)

explicit dependency

The multifrontal QR factorization: StarPU integration

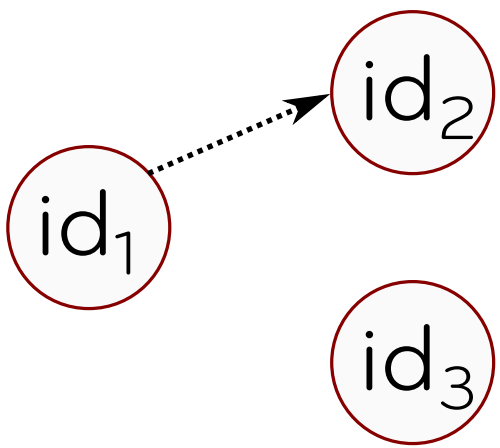
Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: inout, B: in, $id = id_1$)
2: submit_task(fun_2, A: inout, $id = id_2$)

4: submit_task(fun_1, C: inout, D: in, $id = id_3$)



detected dependency (data hazard)

explicit dependency

The multifrontal QR factorization: StarPU integration

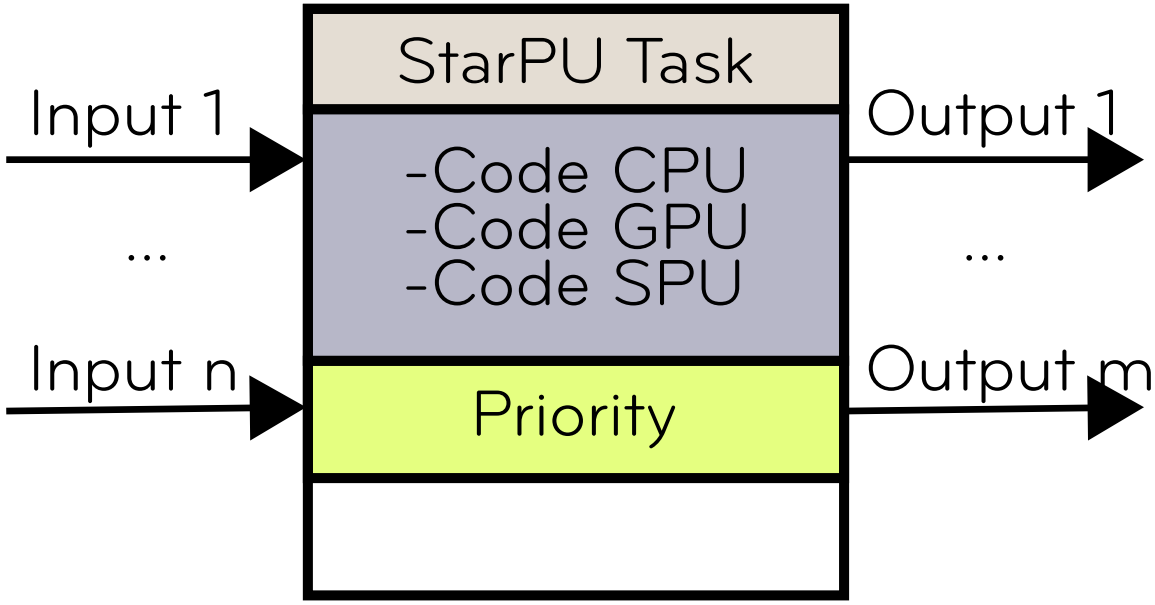
Original sequence:

1: fun_1(A: inout, B: in)
2: fun_2(A: inout)
3: fun_1(C: inout, D: in)

Equivalent StarPU code:

1: submit_task(fun_1, A: inout, B: in, $id = id_1$)
2: submit_task(fun_2, A: inout, $id = id_2$)
3: declare_dependency($id_3 \leftarrow id_1$)
4: submit_task(fun_1, C: inout, D: in, $id = id_3$)

The multifrontal QR factorization: StarPU integration



The easy way: replace all the

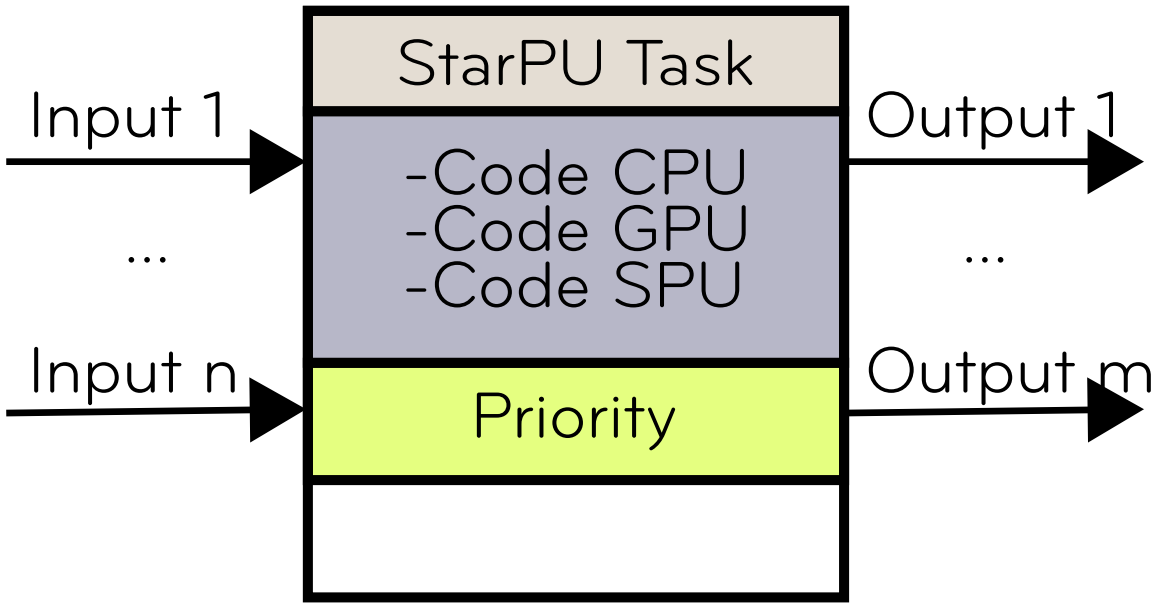
```
call operation1(i1, ..., in, o1, ..., om)
```

with

```
call submit_task(operation1, i1, ..., in, o1, ..., om)
```

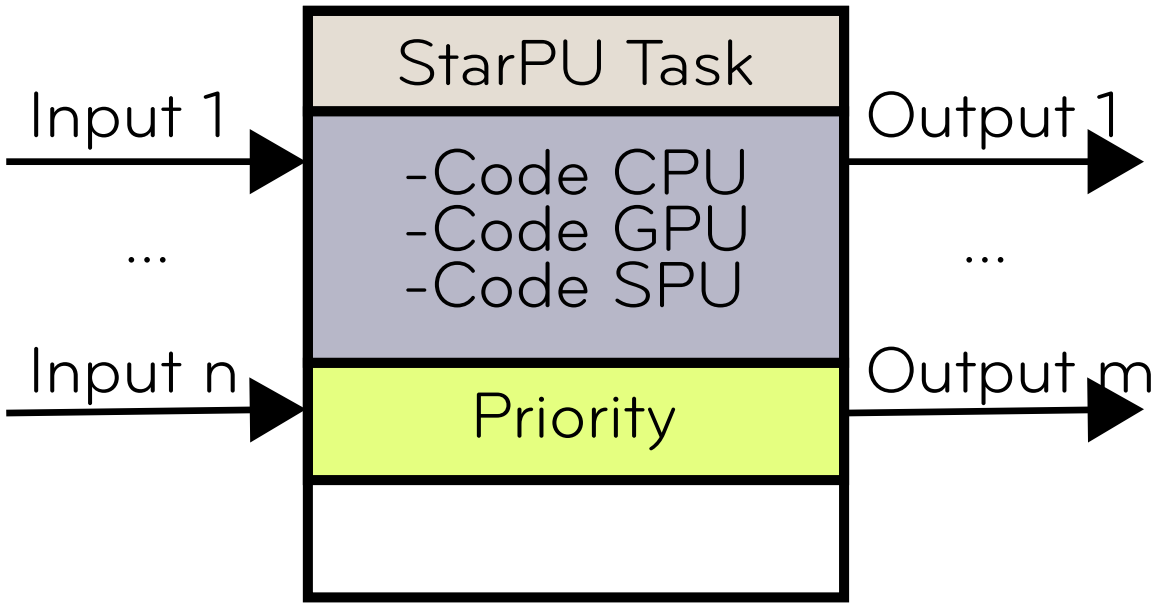
and let StarPU do all the work

The multifrontal QR factorization: StarPU integration



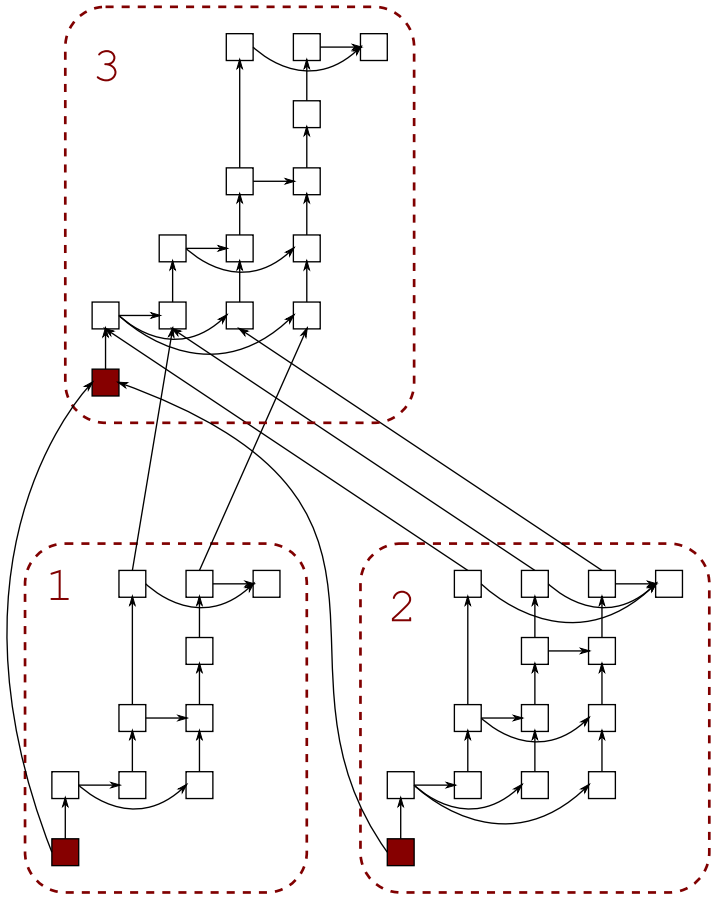
- the DAG may have millions of nodes which makes the scheduling job too complex and memory consuming
- the scheduling of activation tasks have to be controlled in order to limit the memory consumption

The multifrontal QR factorization: StarPU integration



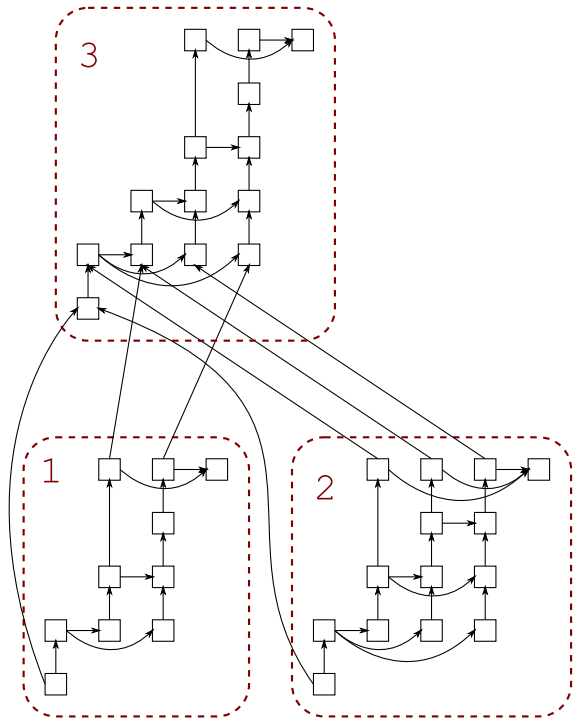
Our approach: We give to StarPU a limited view of the DAG; this is achieved by defining tasks that submit other tasks.

The activation tasks in charge of allocating the memory and preparing the data structures needed for processing a front are the ideal candidates to submit the numerical tasks



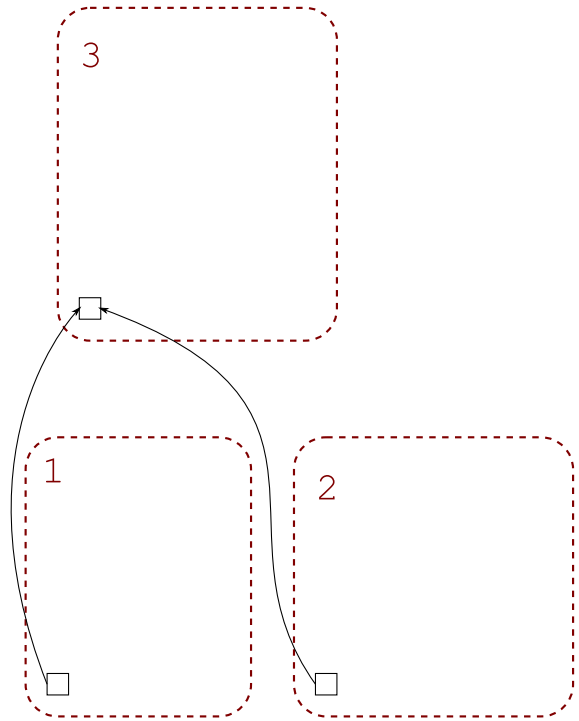
The multifrontal QR factorization: dynamic construction of the DAG

- all the activation tasks are submitted at once with explicit dependencies and a very low priority; this is done in order to prevent StarPU from executing all the activations first
- Upon activation, the numerical tasks corresponding to the activated front are submitted with higher priorities depending on their position in the DAG
- The DAG is progressively unrolled during the factorization and therefore the DAG size is limited as well as the memory consumption



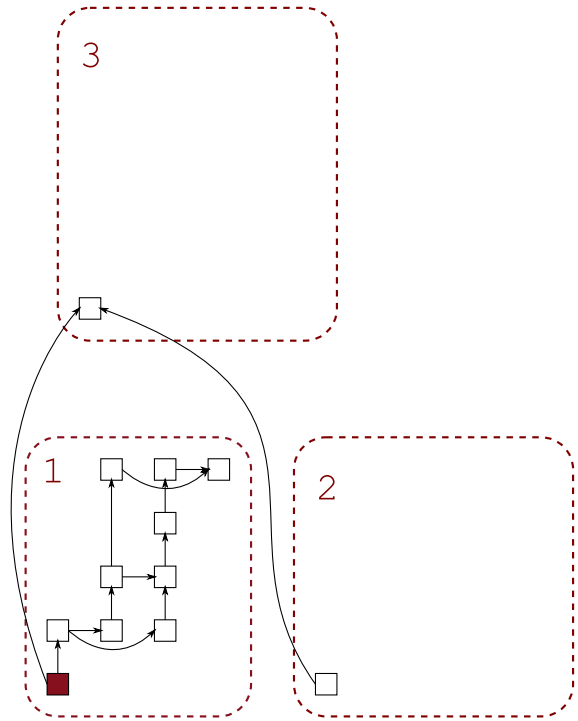
The multifrontal QR factorization: dynamic construction of the DAG

- all the activation tasks are submitted at once with explicit dependencies and a very low priority; this is done in order to prevent StarPU from executing all the activations first
- Upon activation, the numerical tasks corresponding to the activated front are submitted with higher priorities depending on their position in the DAG
- The DAG is progressively unrolled during the factorization and therefore the DAG size is limited as well as the memory consumption



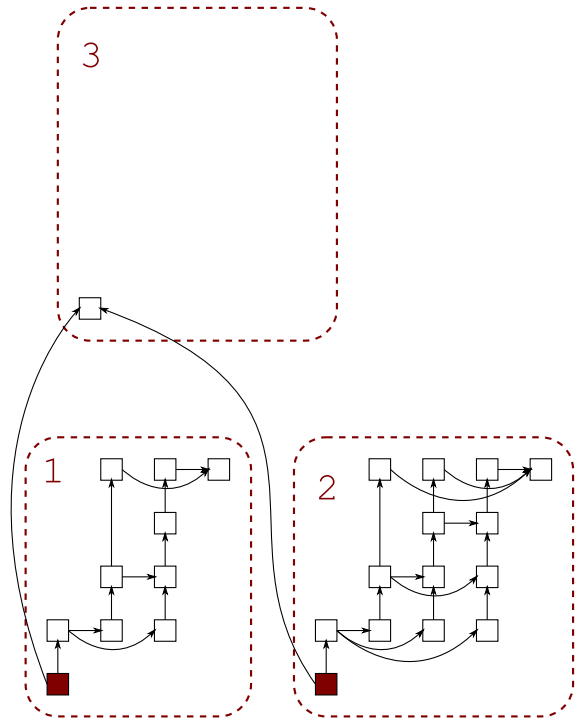
The multifrontal QR factorization: dynamic construction of the DAG

- all the activation tasks are submitted at once with explicit dependencies and a very low priority; this is done in order to prevent StarPU from executing all the activations first
- Upon activation, the numerical tasks corresponding to the activated front are submitted with higher priorities depending on their position in the DAG
- The DAG is progressively unrolled during the factorization and therefore the DAG size is limited as well as the memory consumption



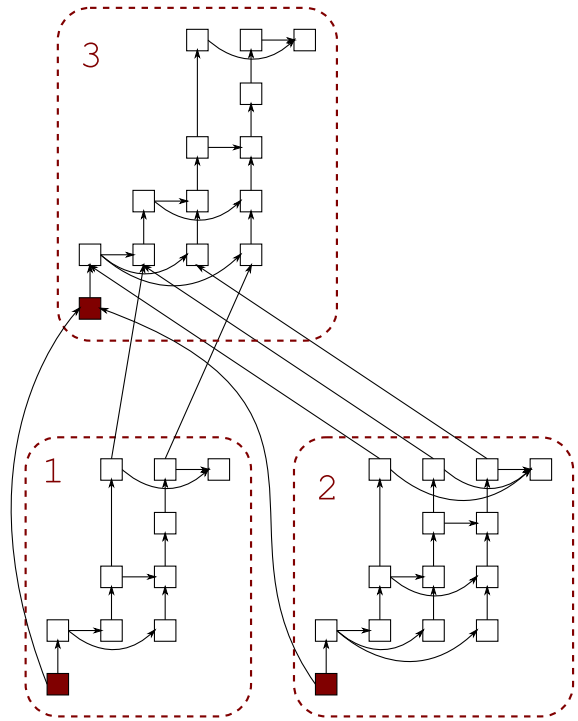
The multifrontal QR factorization: dynamic construction of the DAG

- all the activation tasks are submitted at once with explicit dependencies and a very low priority; this is done in order to prevent StarPU from executing all the activations first
- Upon activation, the numerical tasks corresponding to the activated front are submitted with higher priorities depending on their position in the DAG
- The DAG is progressively unrolled during the factorization and therefore the DAG size is limited as well as the memory consumption



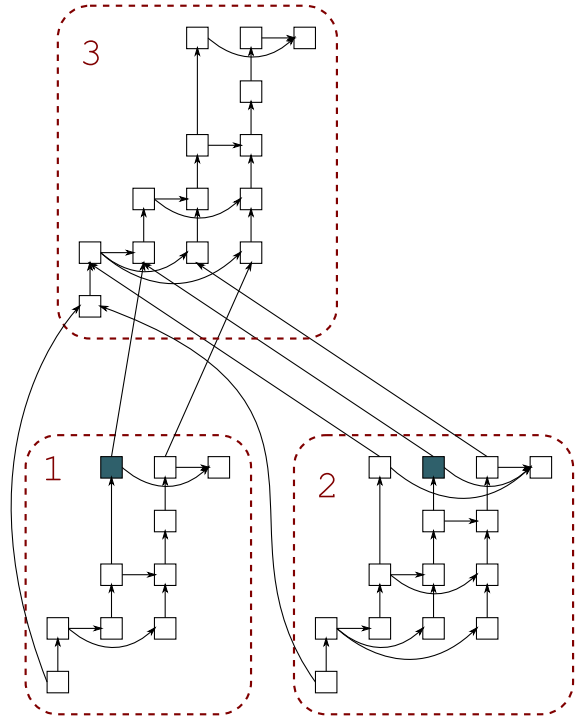
The multifrontal QR factorization: dynamic construction of the DAG

- all the activation tasks are submitted at once with explicit dependencies and a very low priority; this is done in order to prevent StarPU from executing all the activations first
- Upon activation, the numerical tasks corresponding to the activated front are submitted with higher priorities depending on their position in the DAG
- The DAG is progressively unrolled during the factorization and therefore the DAG size is limited as well as the memory consumption



The multifrontal QR factorization: dynamic construction of the DAG

- less eager than the `qr_mumps` scheduler
- the assembly tasks are serialized and are executed in order of submission although it is unnecessary. Need a commute flag in StarPU for commutative operations

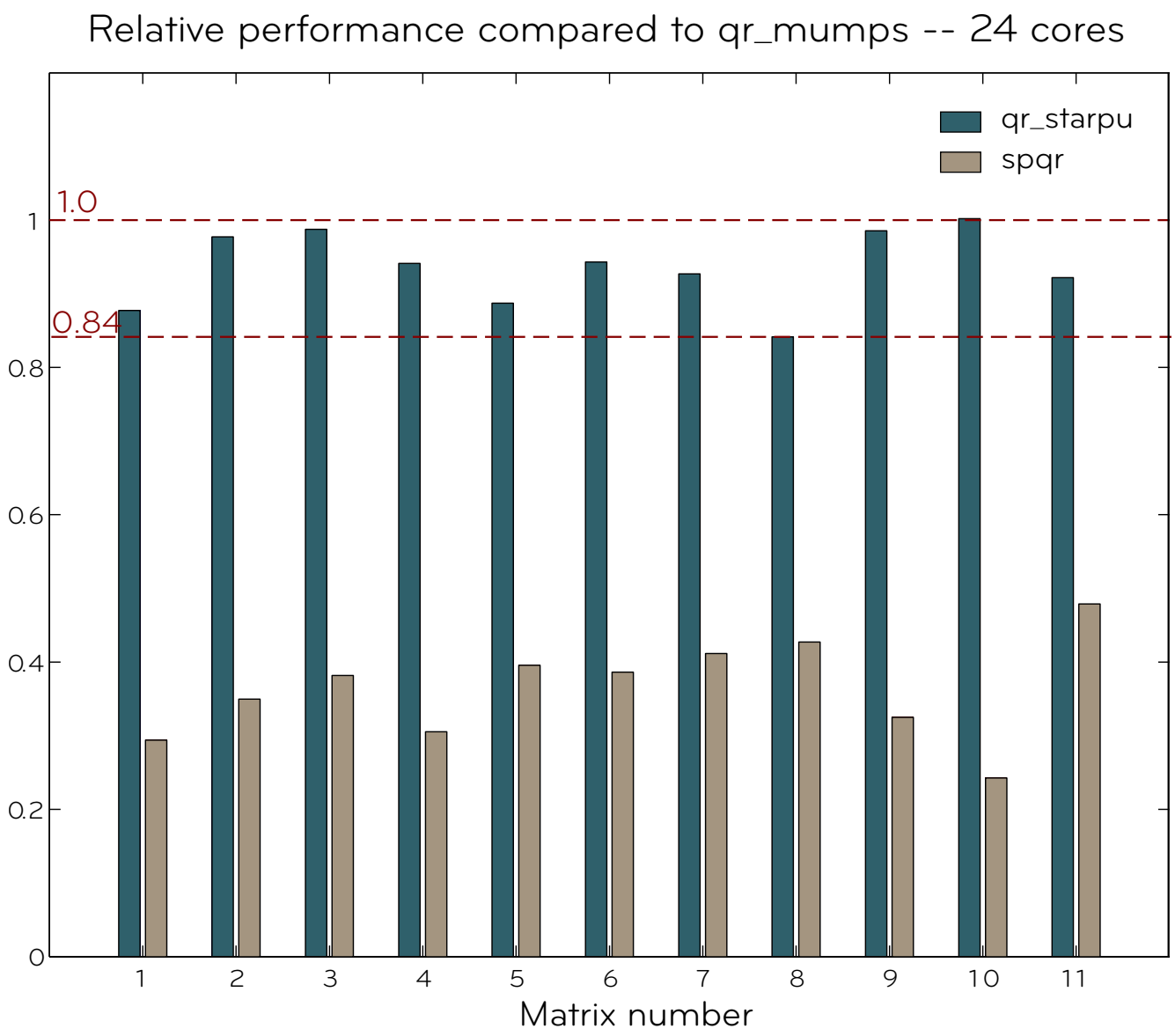


Experimental setup

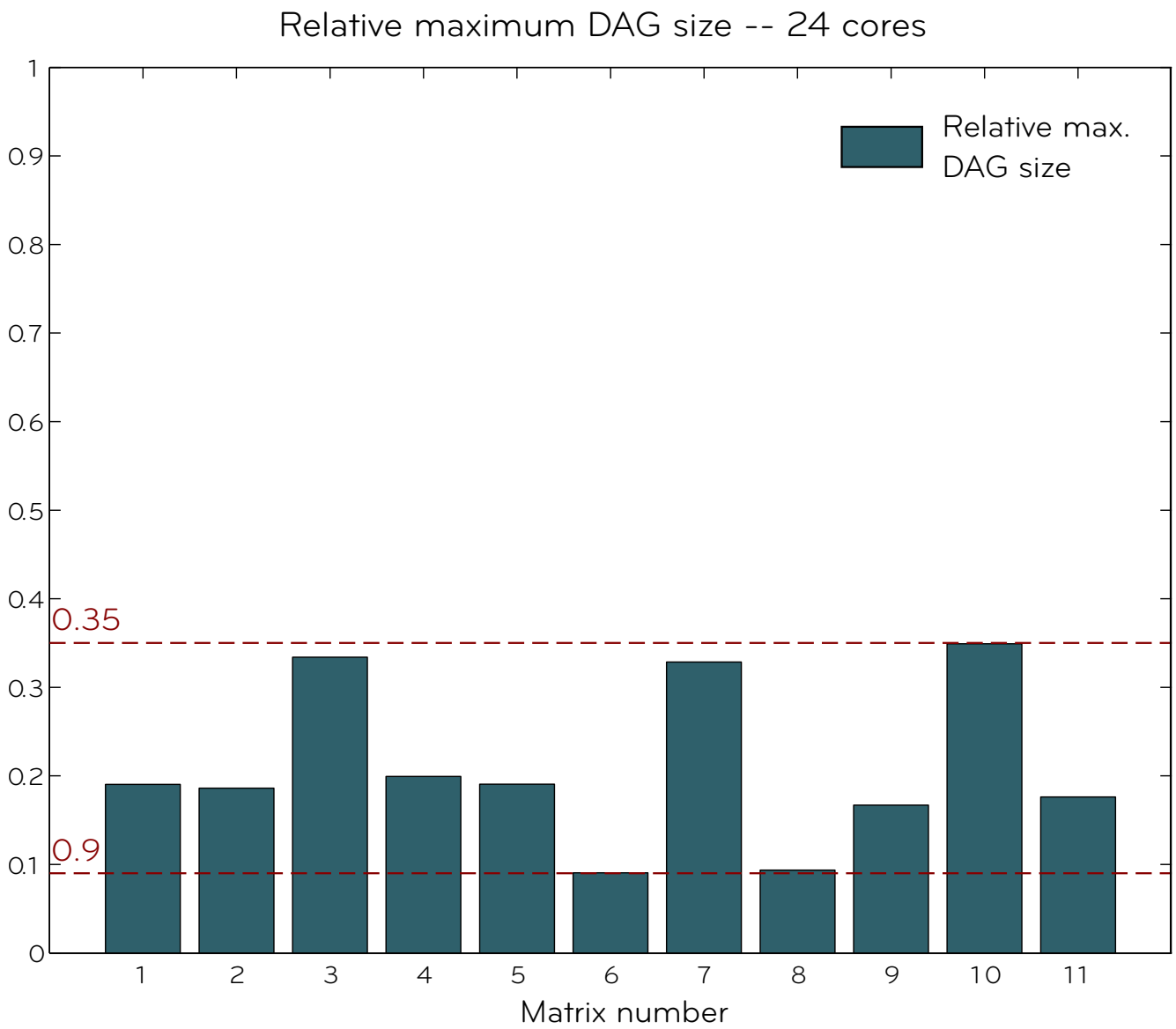
- Platform:
  - 4× AMD hexacore
  - 76 GB of memory (in 4 NUMA modules)
  - GNU 4.4 compilers
  - MKL 10.2
- Problems: a set of matrices from the UF collection

#	Matrix	m	n	nnz	flops
1	tp-6	142752	1014301	11537419	255 G
2	karted	46502	133115	1770349	258 G
3	EternityII_E	11077	262144	1572792	544 G
4	degme	185501	659415	8127528	591 G
5	cat_ears_4_4	19020	44448	132888	716 G
6	Hirlam	1385270	452200	2713200	2401 G
7	e18	24617	38602	156466	3399 G
8	flower_7_4	27693	67593	202218	4261 G
9	Rucci1	1977885	109900	7791168	12768 G
10	sls	1748122	62729	6804304	22716 G
11	TF17	38132	48630	586218	38209 G

Experimental results



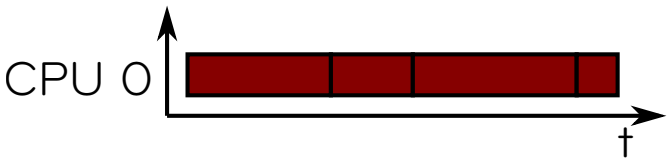
Experimental results



Experimental results

Sequential execution

Parallel execution



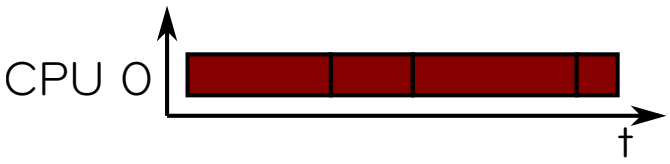
Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$

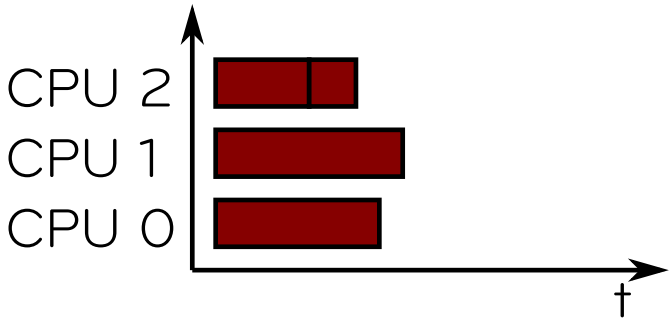
Parallel efficiency $e(p)$ (p processors):

Experimental results

Sequential execution



Parallel execution



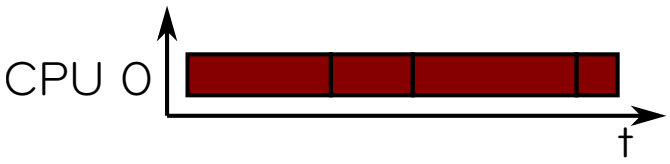
Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$
- $t_c(p)$: task time

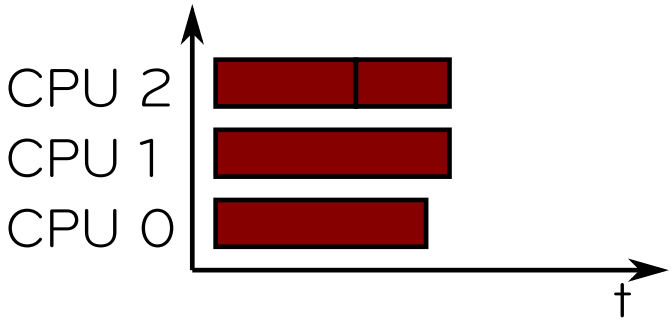
Parallel efficiency $e(p)$ (p processors):

Experimental results

Sequential execution



Parallel execution



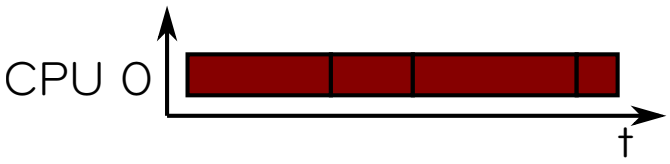
Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$
- $t_c(p)$: task time ($\neq t_c(1)$)

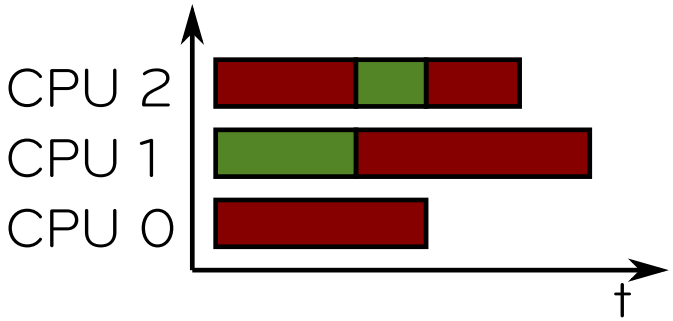
Parallel efficiency $e(p)$ (p processors):

Experimental results

Sequential execution



Parallel execution



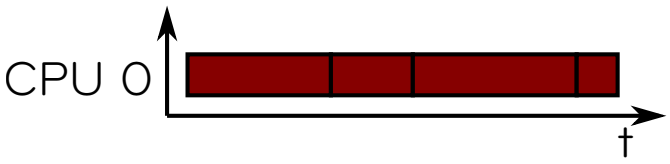
Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$
- $t_c(p)$: task time ($\neq t_c(1)$)
- $t_i(p)$: idle time

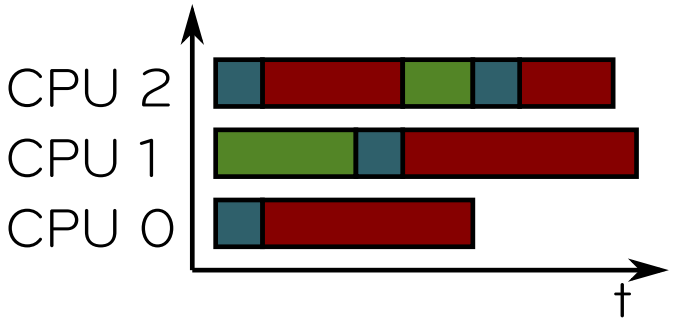
Parallel efficiency $e(p)$ (p processors):

Experimental results

Sequential execution



Parallel execution



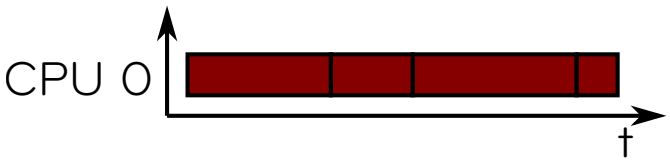
Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$
- $t_c(p)$: task time ($\neq t_c(1)$)
- $t_i(p)$: idle time
- $t_s(p)$: scheduler time

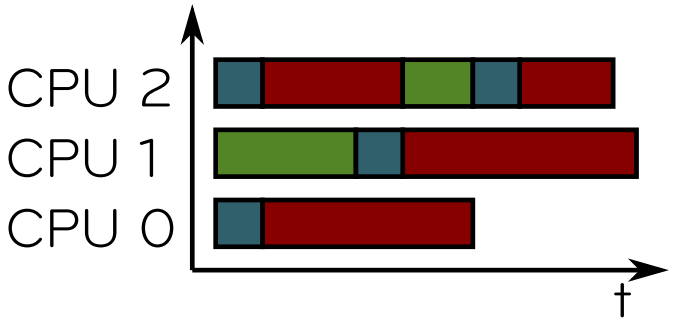
Parallel efficiency $e(p)$ (p processors):

Experimental results

Sequential execution



Parallel execution



Cumulative times (p processors) measured during the factorization:

- reference sequential time $t_c(1)$
- $t_c(p)$: task time ($\neq t_c(1)$)
- $t_i(p)$: idle time
- $t_s(p)$: scheduler time

Parallel efficiency $e(p)$ (p processors):

$$e(p) = \frac{t_c(1)}{t_c(p) + t_s(p) + t_i(p)}$$

Experimental results

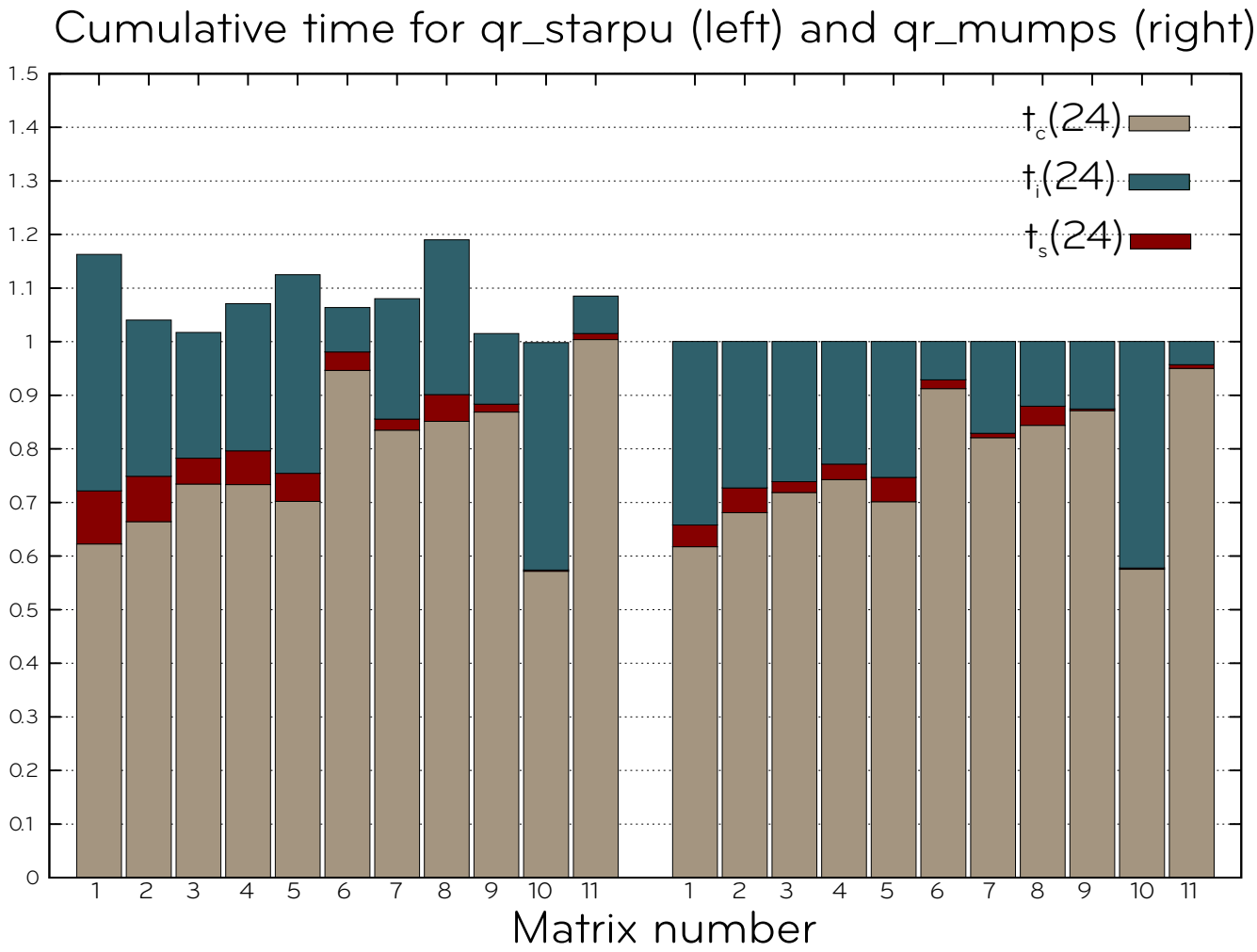
Efficiency of the parallelism $e(p)$ (p processors) decomposed into three efficiencies:

$$e(p) = \overbrace{\frac{t_c(1)}{t_c(p)}}^{e_l} \cdot \overbrace{\frac{t_c(p) + t_s(p)}{t_c(p) + t_s(p) + t_i(p)}}^{e_p} \cdot \overbrace{\frac{t_c(p)}{t_c(p) + t_s(p)}}^{e_s}$$

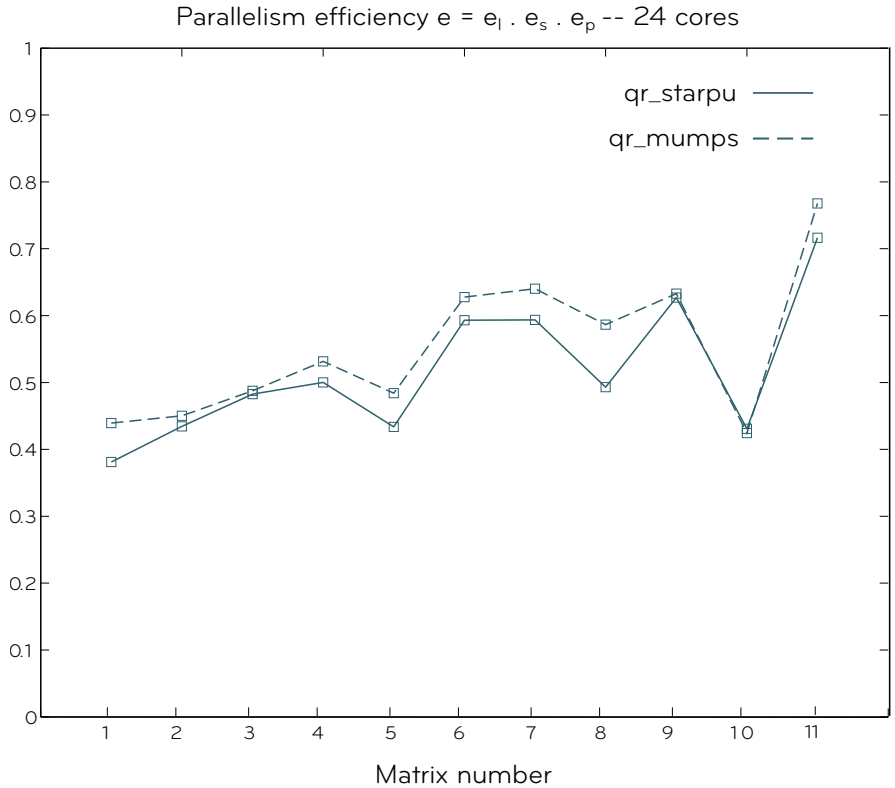
These efficiencies correspond to the three identified effects:

- $e_l(p)$: locality efficiency
- $e_p(p)$: pipeline efficiency
- $e_s(p)$: scheduler efficiency

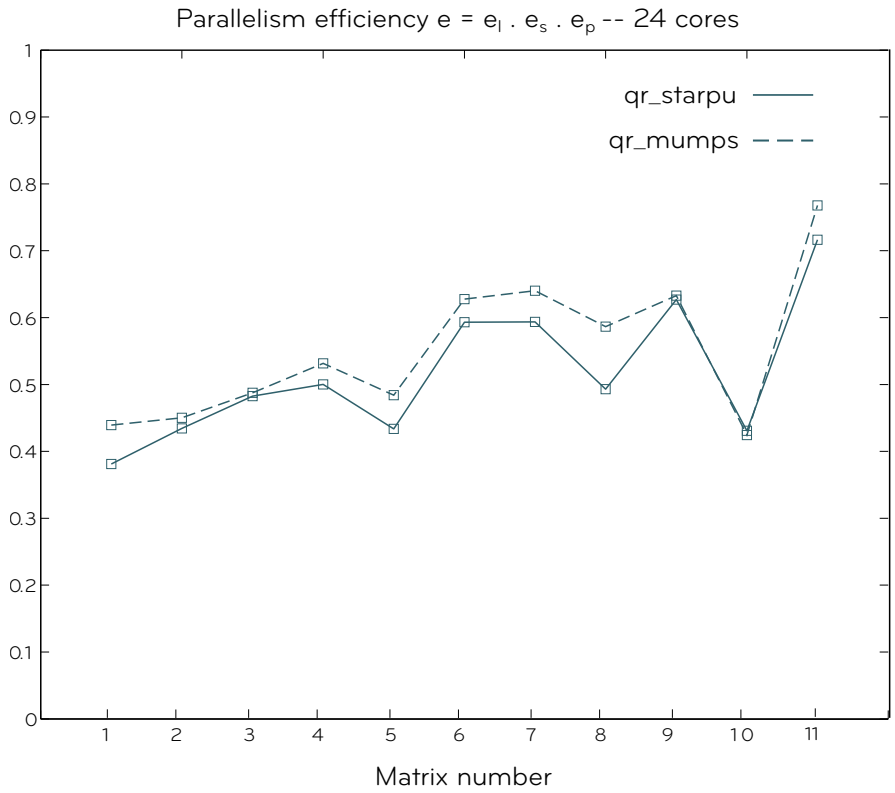
Experimental results

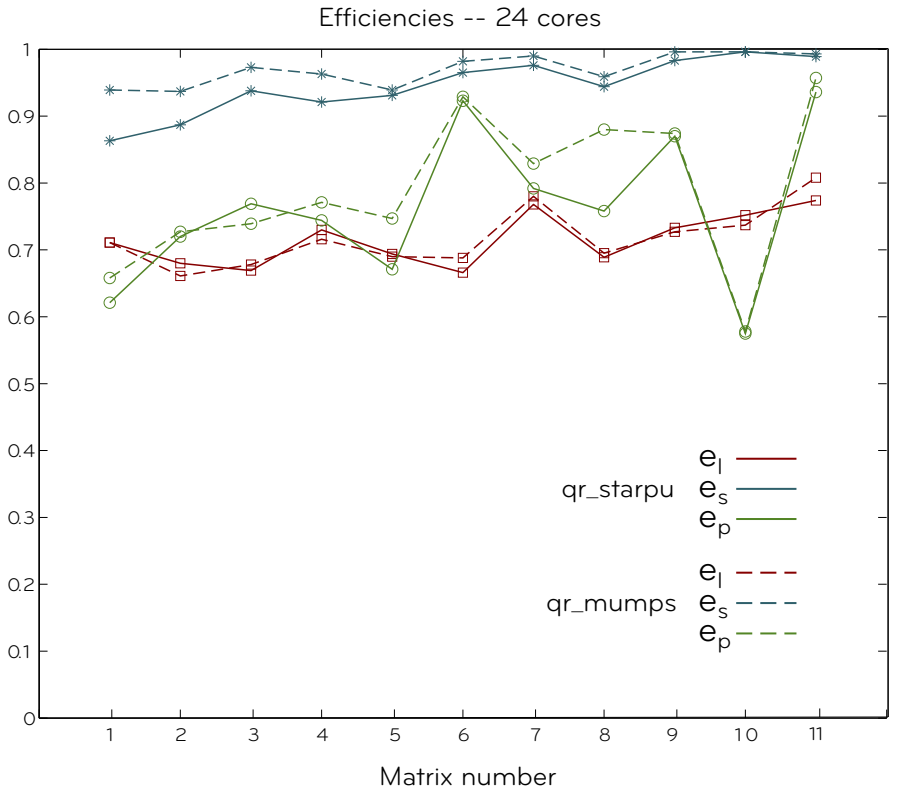


Experimental results



Experimental results

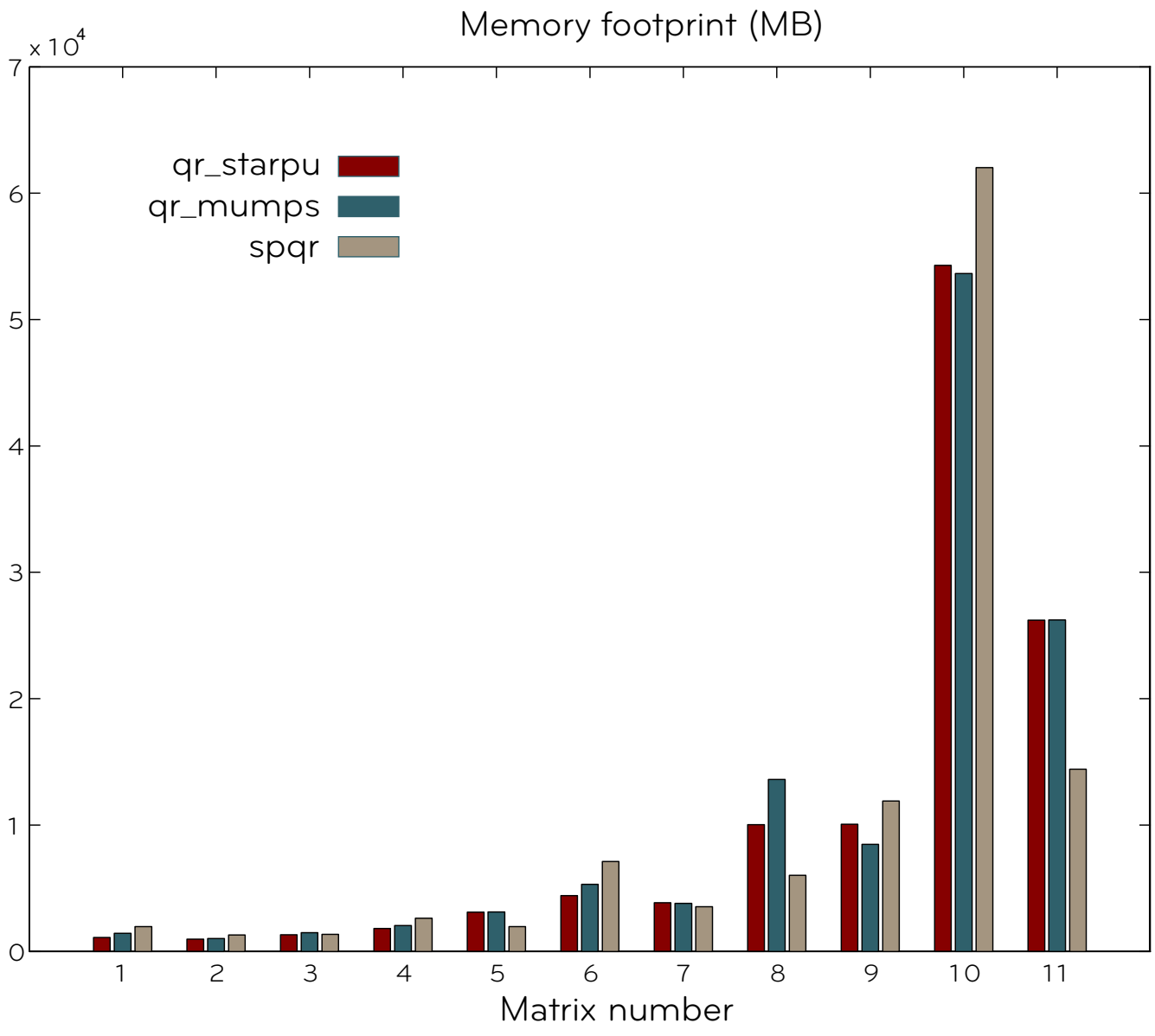




- similar locality efficiencies for both codes
- higher scheduling overhead for small matrices especially with `qr_starpu`

- poorer pipeline for `qr_starpu` mainly due to:
  - the serialization of assembly tasks
  - the strategy of front activation limiting the parallelism

Experimental results



Experimental results: conclusion

- `qr_starpu` achieves a very competitive performance compared to `qr_mumps` and an excellent memory behaviour
- higher but still marginal scheduling overhead imposed by the runtime system
- good scalability of the runtime system thanks to the adaptive task submission
- minor limitations of StarPU common to many other runtime system environments

This version constitutes a good basis for the development of a multifrontal method for heterogeneous architectures

Ongoing & future work

Ongoing work

- possibility to declare commutative operations in StarPU
- Use GPUs to process the biggest fronts (top of the tree)
- consider other models to express the task graph: Compare the Sequential Task Flow model (STF) of StarPU with the Parametrized Task graph (PTG) model of PaRSEC (collaboration with George Bosilca)
- control the memory consumption with a memory-aware algorithm (shared-memory version of the approach presented by François-Henry Rouet)
- use of scheduling contexts in StarPU (Andra Hugo et al.) to achieve a better data locality.

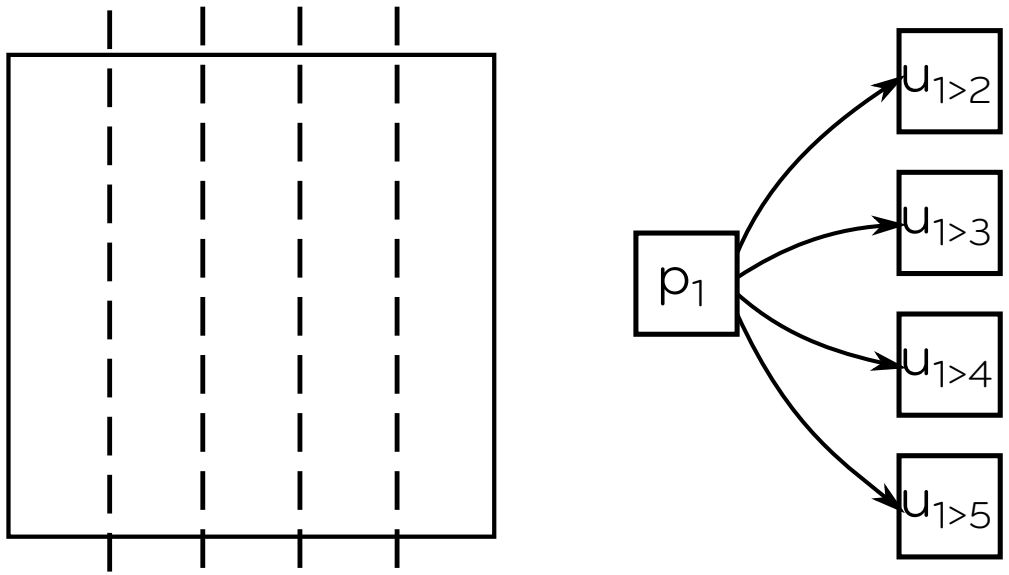
Future work

- scheduling strategies (use of performance models)
- process the factorization of entire sub-trees on GPUs (potential collaboration with T. Davis)
- 2D front partitioning
- MIC architectures

Thanks!
Questions?

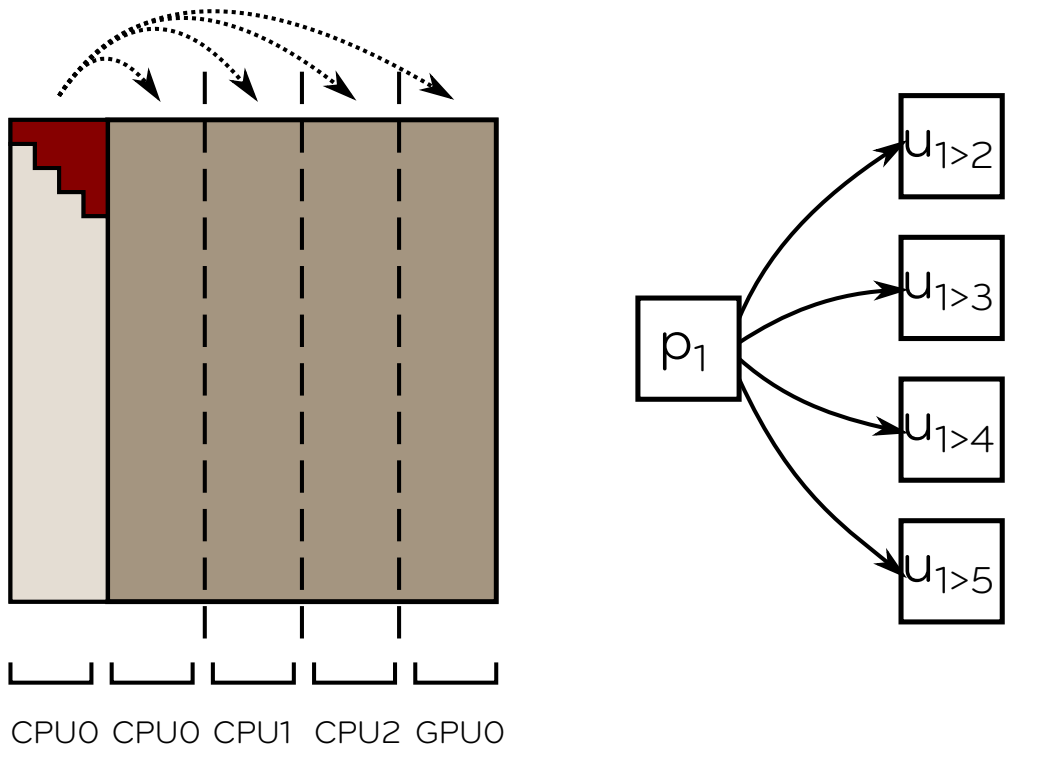
Ongoing work

Use GPUs to process the biggest fronts (top of the tree).
MAGMA-like approch: panel operations on CPUs and update on GPUs



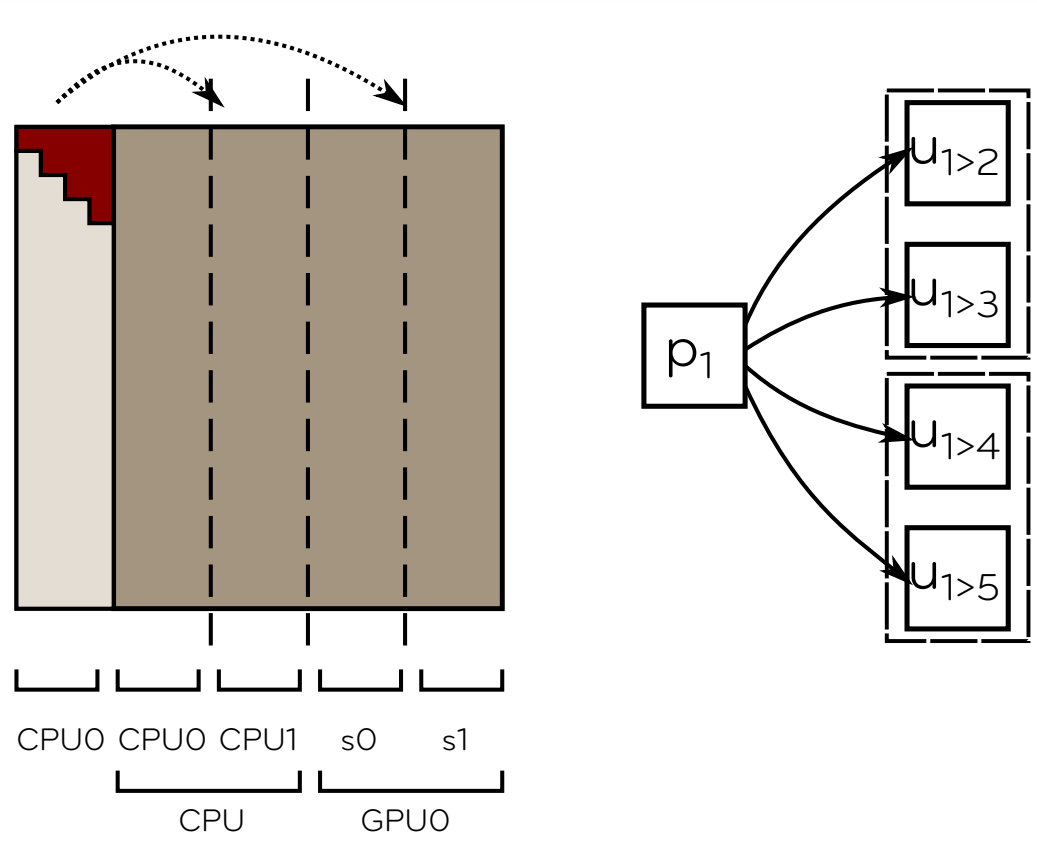
Ongoing work

Use GPUs to process the biggest fronts (top of the tree).
MAGMA-like approch: panel operations on CPUs and update on GPUs



Ongoing work

Use GPUs to process the biggest fronts (top of the tree).
MAGMA-like approch: panel operations on CPUs and update on GPUs



Ongoing work