Direct methods on GPU-based systems

Preliminary work towards a functioning code

A. Decollas and F. Lopez, Joint work with IRIT Toulouse, LaBRI / Inria Bordeaux, LIP / Inria Lyon

Sparse Days 2012. Toulouse, June 25th

Context of the work

Context

F. Lopez @ IRIT-Toulouse

Evaluate the efficiency of modern runtime systems for heterogeneous and irregular workloads such as Multifrontal solvers on homogeneous, multicore architectures.

A. Decollas @ Inria-Bordeaux

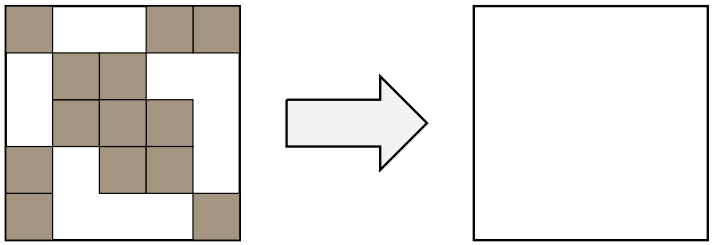
Develop dense linear algebra kernels specific to sparse, direct solvers capable of achieving high efficiency on heterogeneous systems equipped with multiple CPUs and GPUs.

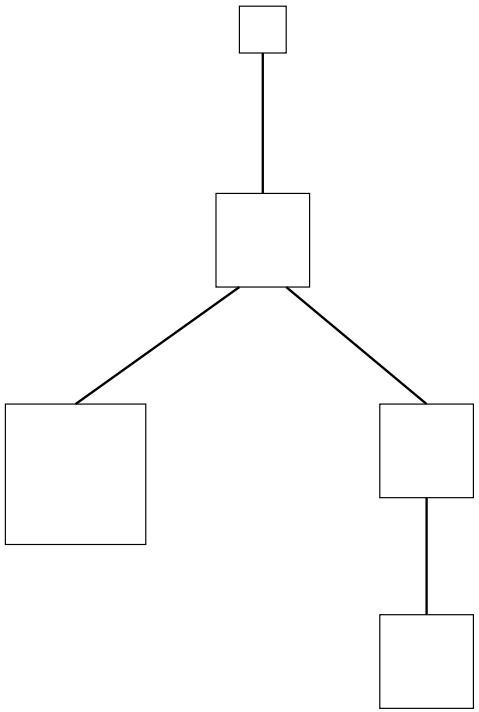
These two activities will ultimately be merged into a sparse, direct solver for accelerated multicore systems.

The multifrontal method

The multifrontal 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 rows/columns related to the pivots and all the other coefficients concerned by their elimination

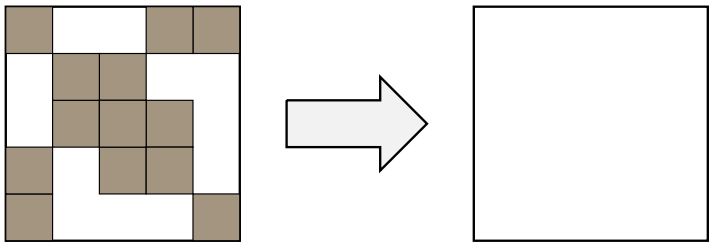


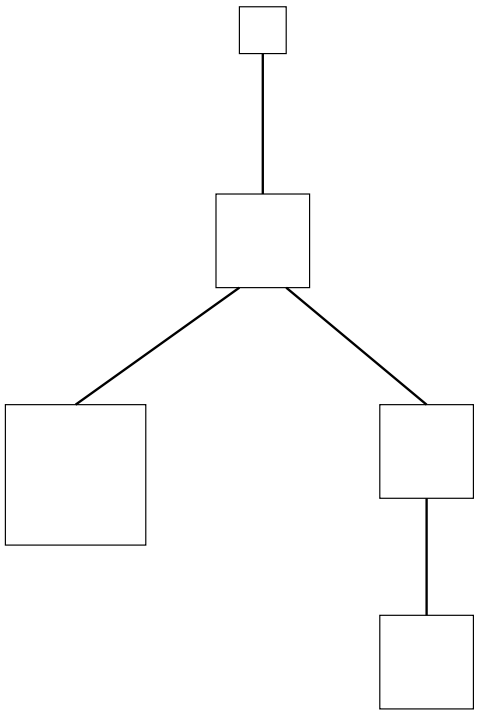


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

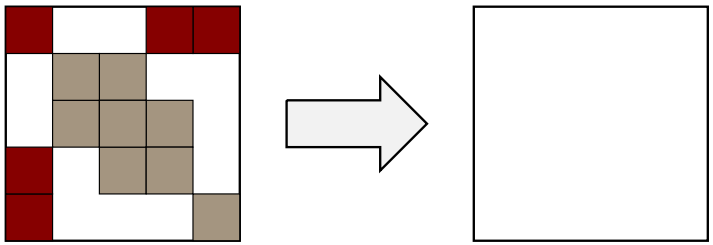


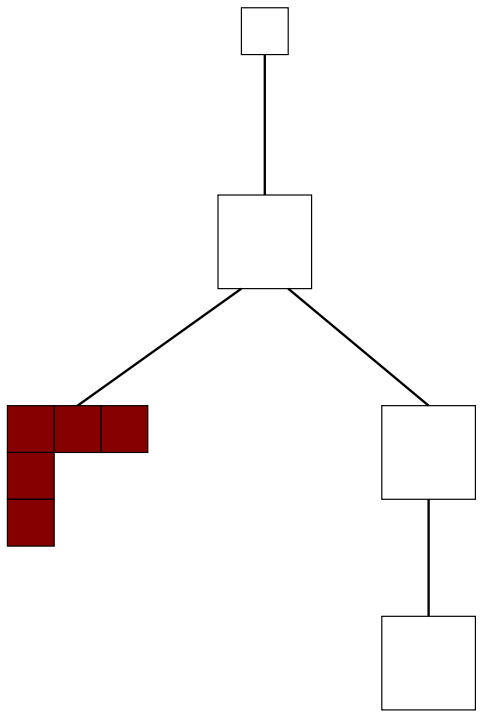


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

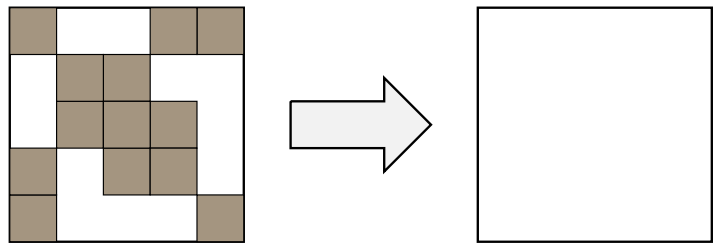


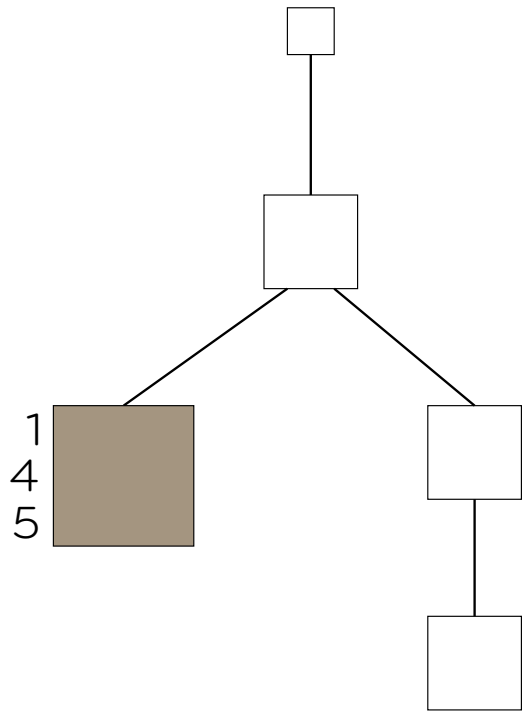


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

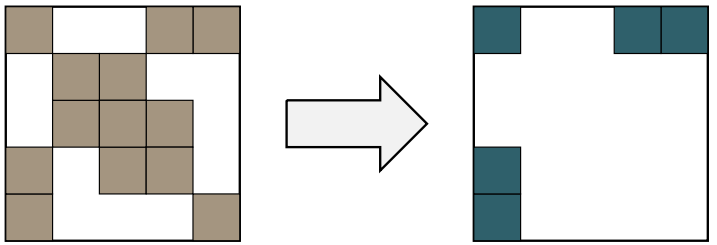


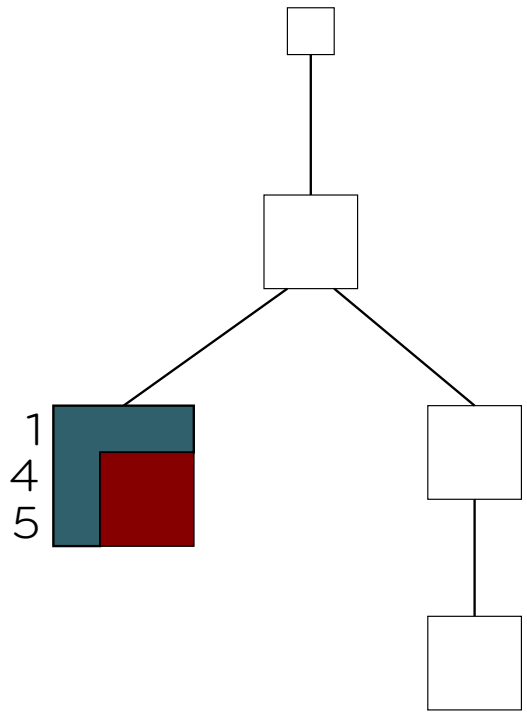


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

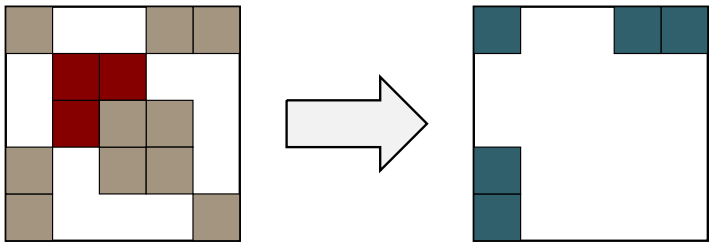


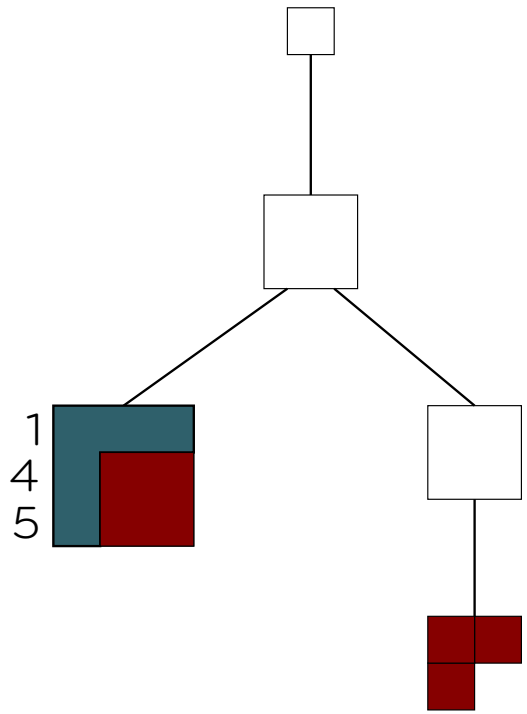


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

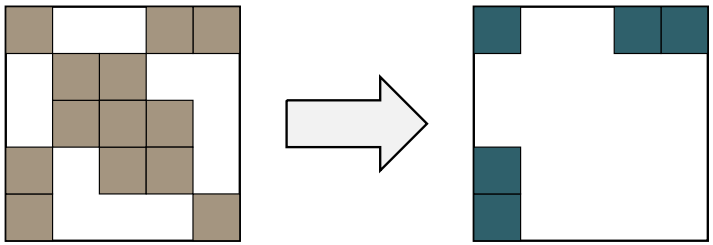


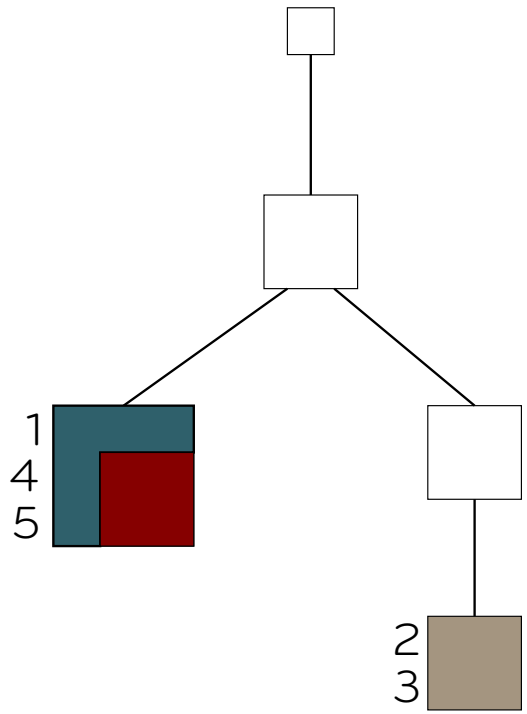


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

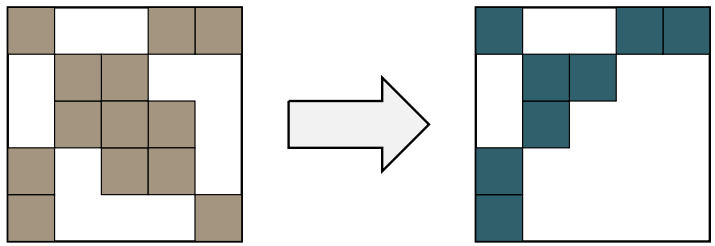


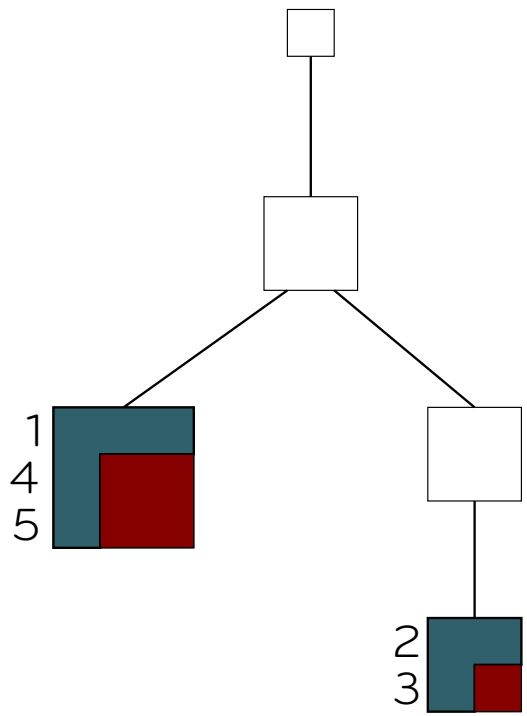


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

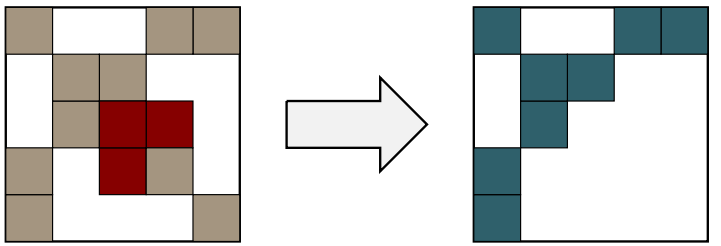


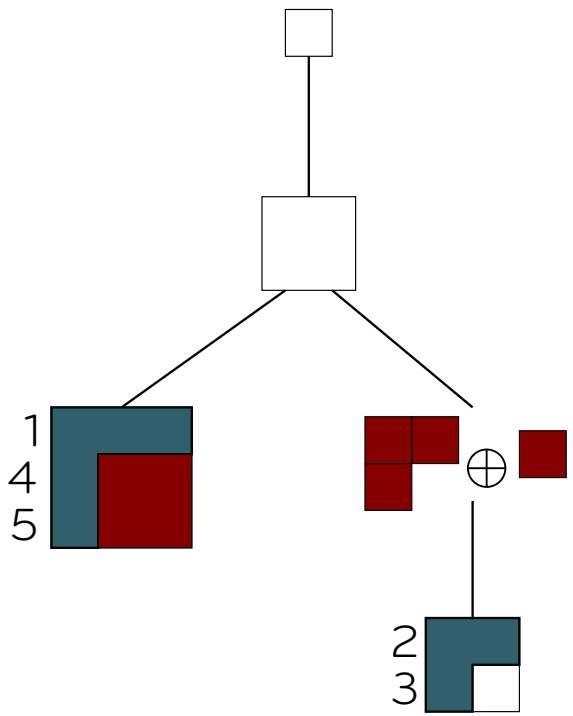


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

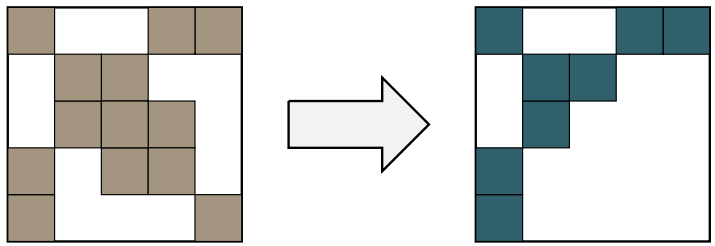


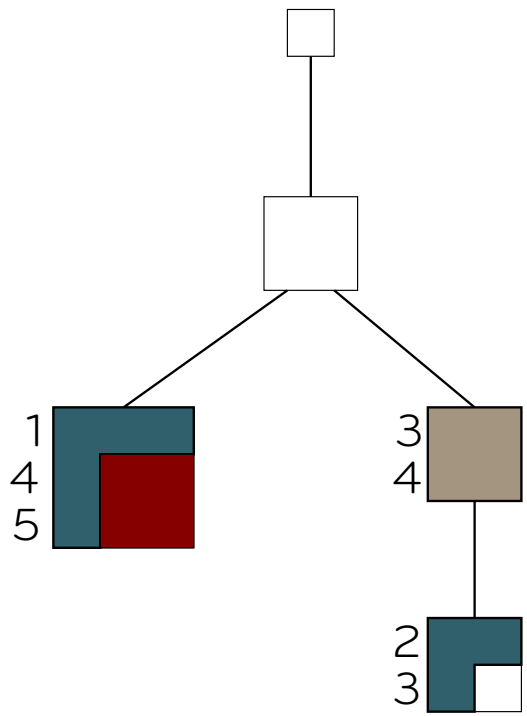


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

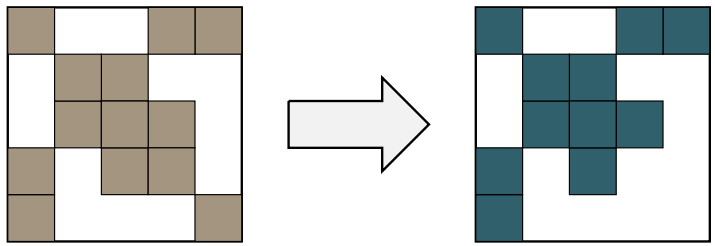


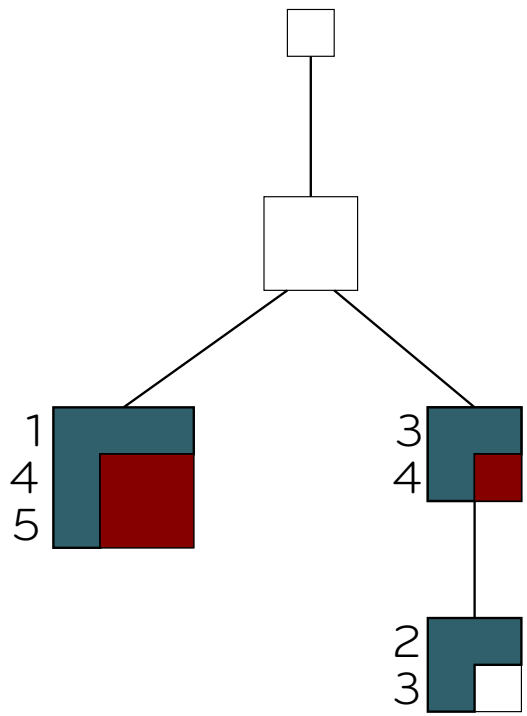


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front

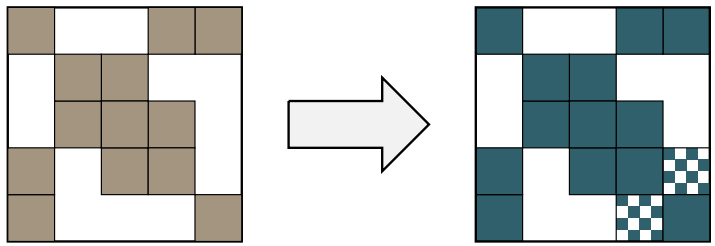


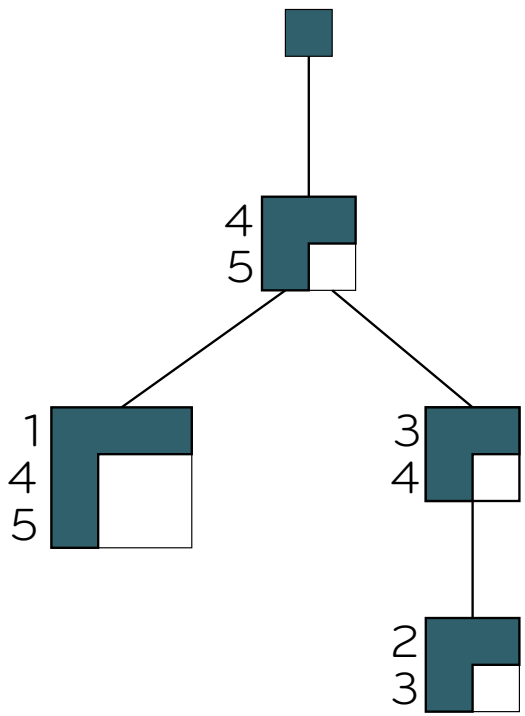


The multifrontal method

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

- assembly: a set of coefficients from the original matrix associated with the pivots and a number of *contribution blocks* produced by the treatment of the child nodes are summed to form the frontal matrix
- factorization: the k pivots are eliminated through a partial factorization of the frontal matrix. As a result we get:
  - k rows/columns of the global factors
  - A *contribution block* that will be assembled into the parent's front





CPU-GPU hybrid architectures

GPUs may be used as powerful accelerators for HPC applications:

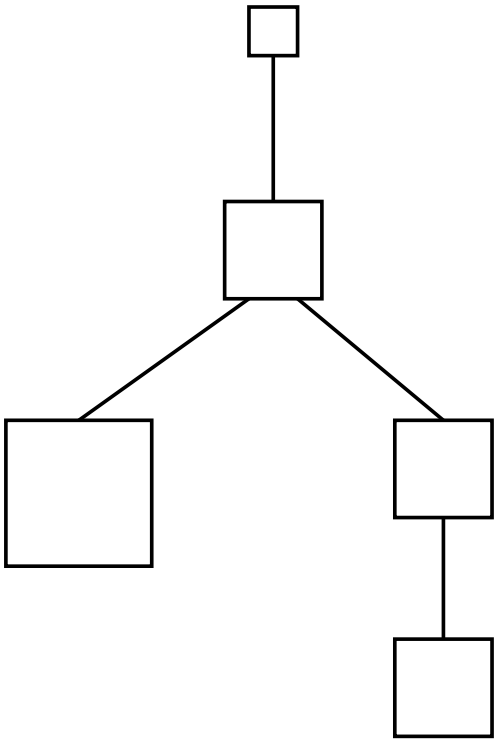
- ▲ High computational performance (comparison GPU-CPU: 10× faster, memory access 5× faster)
- ▲ Energy efficient

despite these capabilities, the use of GPUs is challenging:

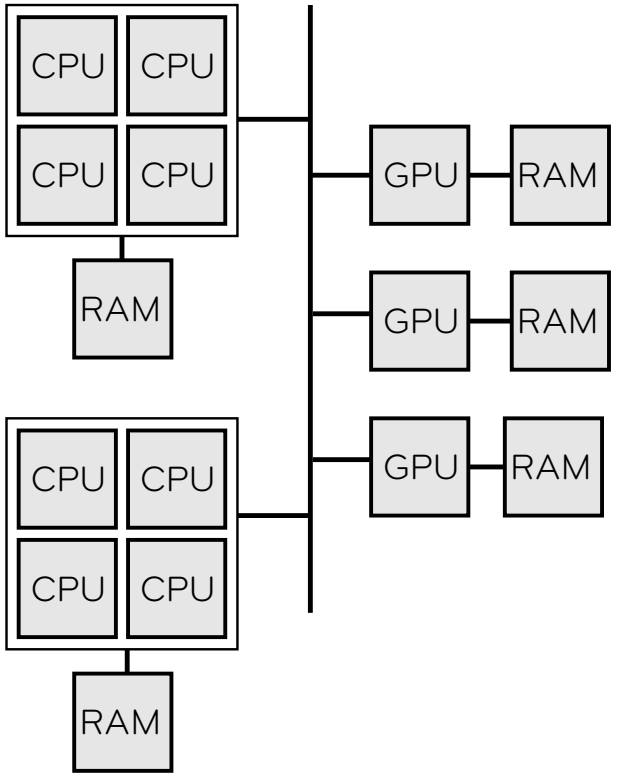
- ▼ Complex architectures (comparison GPU-CPU : 100× more cores)
- ▼ CPU-GPU programming models incompatible.
- ▼ CPU $\leftrightarrow$ GPU transfers are expensive (no shared memory)

$$\Rightarrow \text{specific algorithms}$$

CPU-GPU hybrid architectures



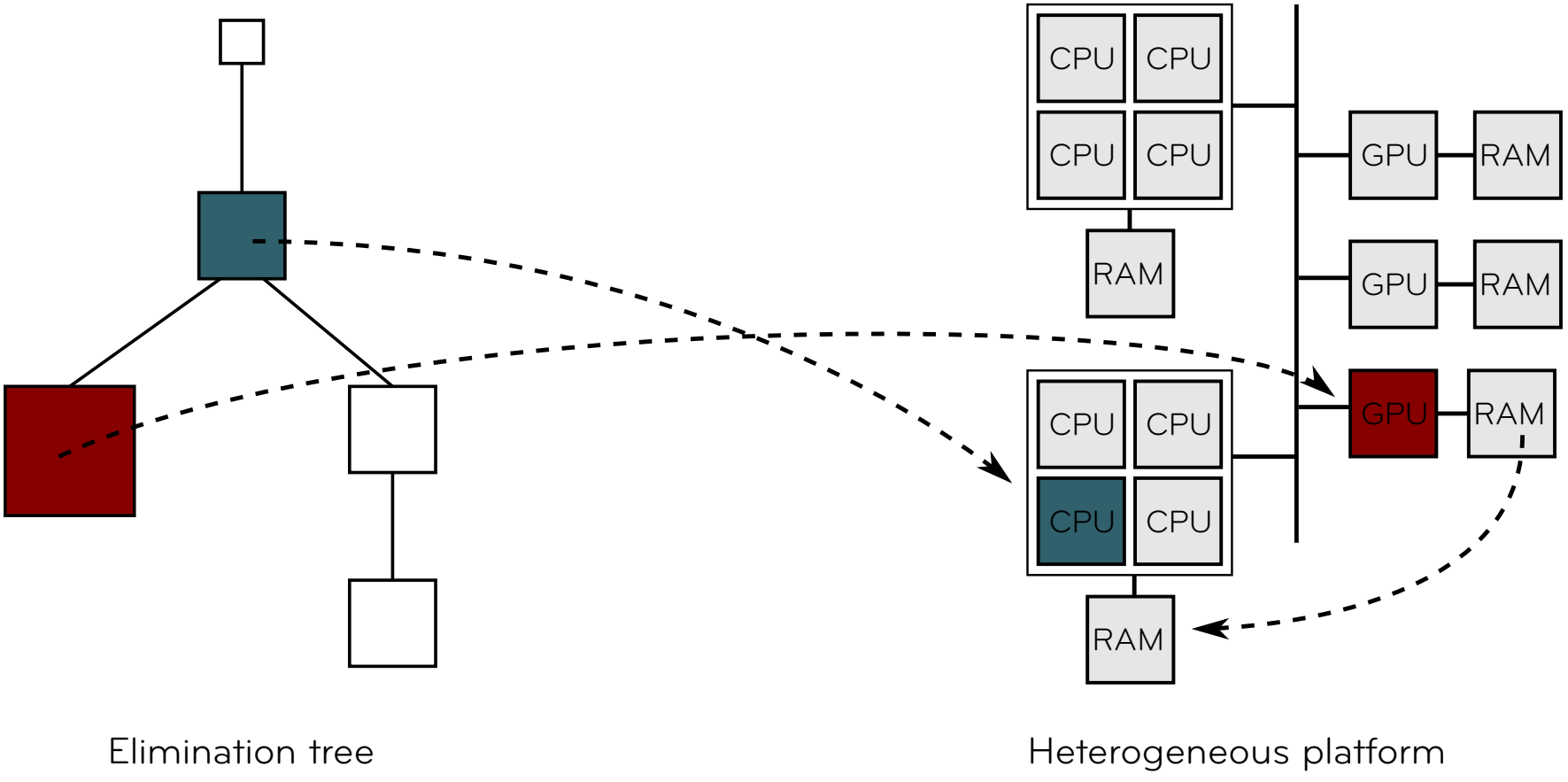
Elimination tree



Heterogeneous platform

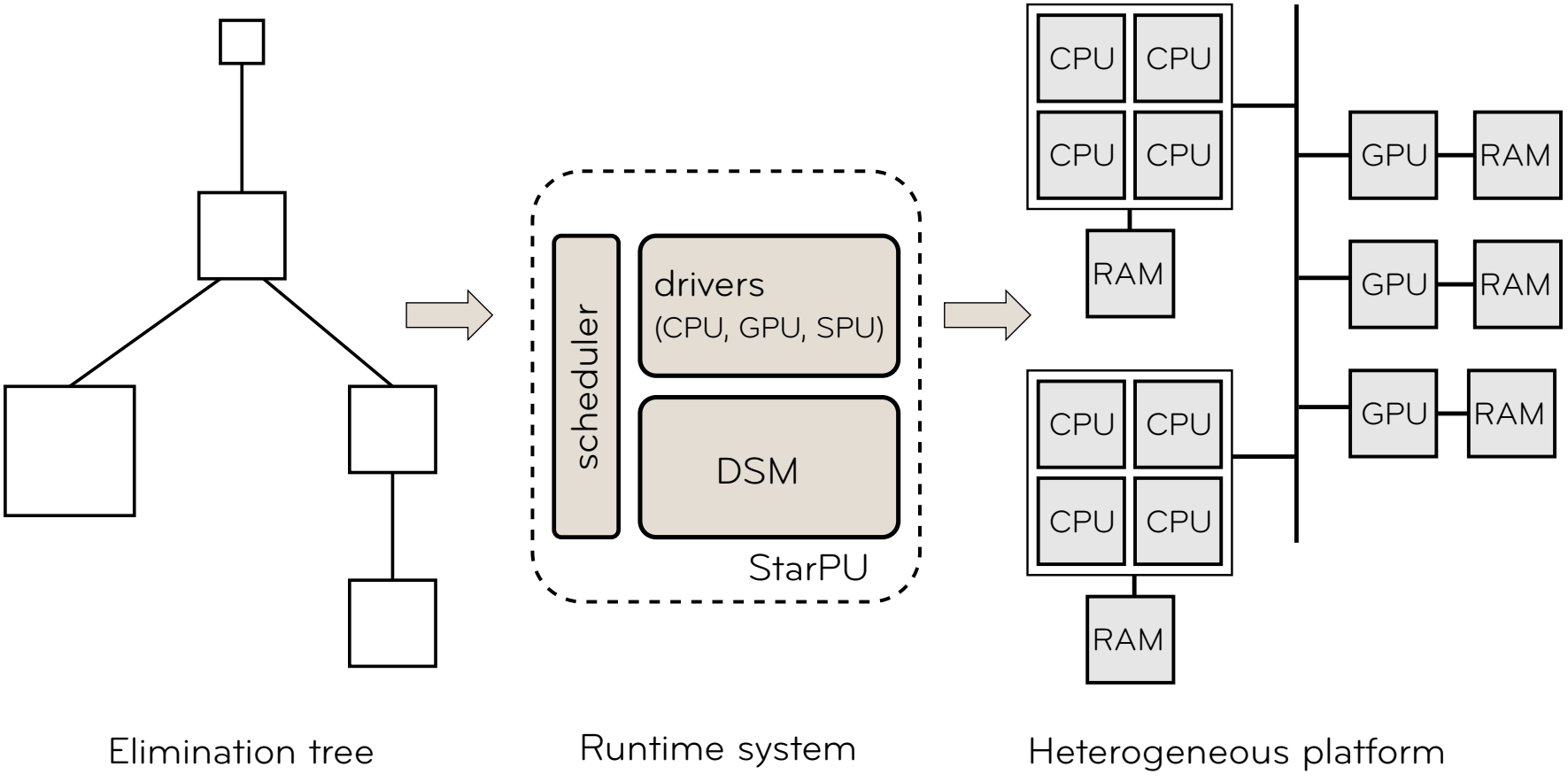
- An extremely heterogeneous workload
- A heterogeneous architecture
- mapping tasks is challenging

CPU-GPU hybrid architectures



One option is to do the mapping by hand (see T. Davis' talk at SIAM PP12). This requires a very accurate performance models difficult to achieve.

CPU-GPU hybrid architectures



Another option is to exploit the features of a modern runtime system capable of handling the scheduling and the data coherency 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)
- Coherency management of manipulated data.

Multifrontal QR factorization on multicores

The multifrontal QR factorization

The multifrontal QR factorization of a sparse matrix $A = QR$ follows the pattern defined by the Cholesky factorization of the associated normal equations $A^T A = LL^T$ because of the equivalence of R and L.

It shares most of the features of the multifrontal Cholesky algorithm apart from (most importantly):

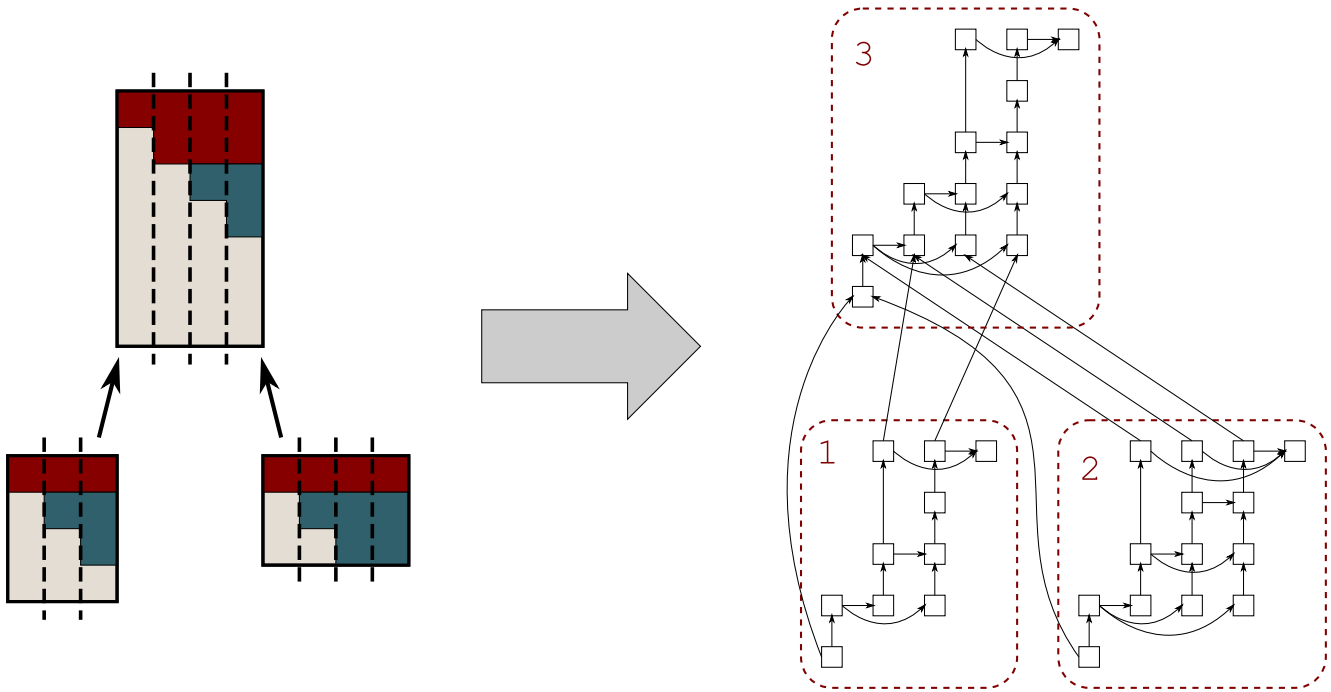
- Frontal matrices are, in general, rectangular (both over or under-determined)
- Frontal matrices are fully factorized
- Contribution blocks are stacked and not summed

The multifrontal QR factorization: parallelism

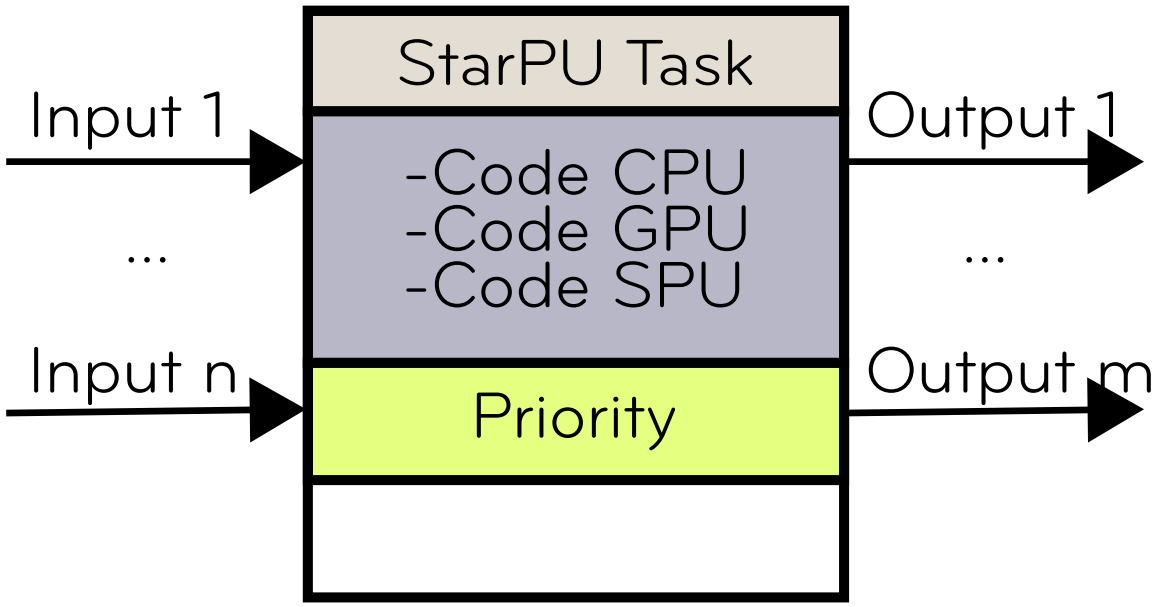
Parallelism comes from two sources:

- Tree: nodes in separate branches can be treated independently
- Node: large nodes can be treated by multiple processes

In `qr_mumps` both sources are exploited consistently, by partitioning the frontal matrices and replacing the elimination tree with a DAG:

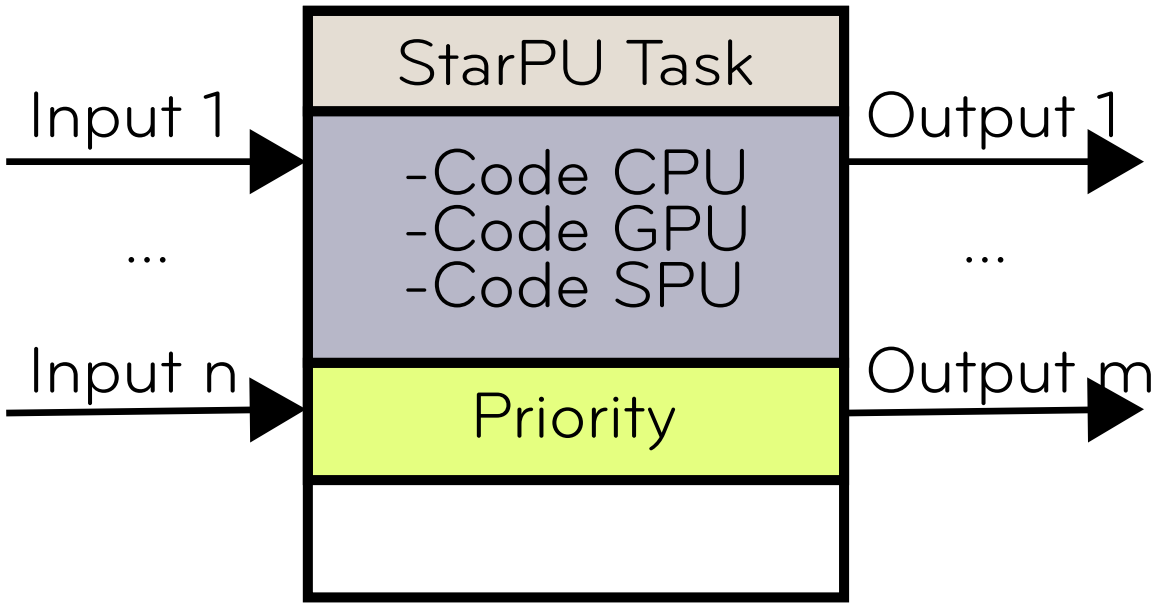


The multifrontal QR factorization: StarPU integration



- 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



The easy way: replace all the

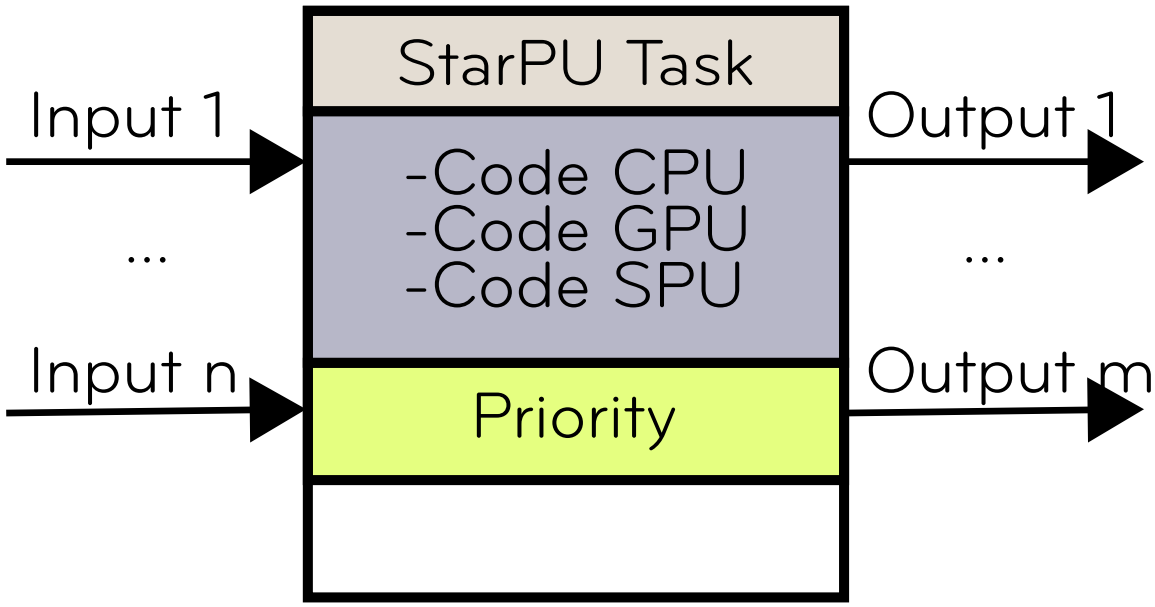
```
call operation1(i1, ..., in, o1, ..., om)
```

with

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

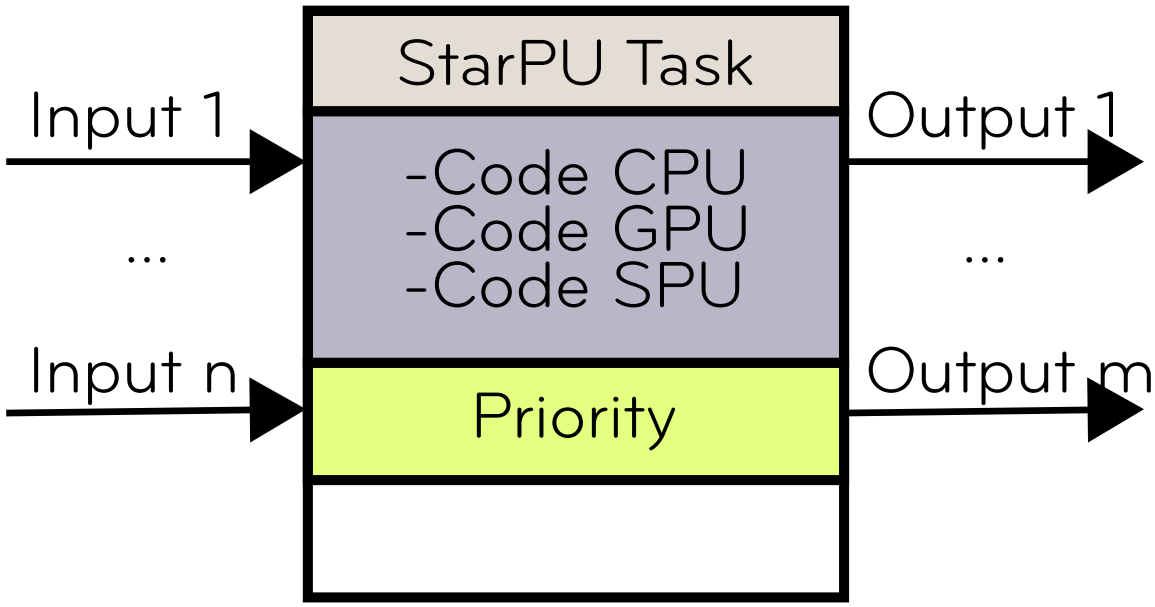
and let StarPU do all the work

The multifrontal QR factorization: StarPU integration



This is functionally correct but the DAG may have millions of nodes which makes the scheduling job too complex and memory consuming

The multifrontal QR factorization: StarPU integration

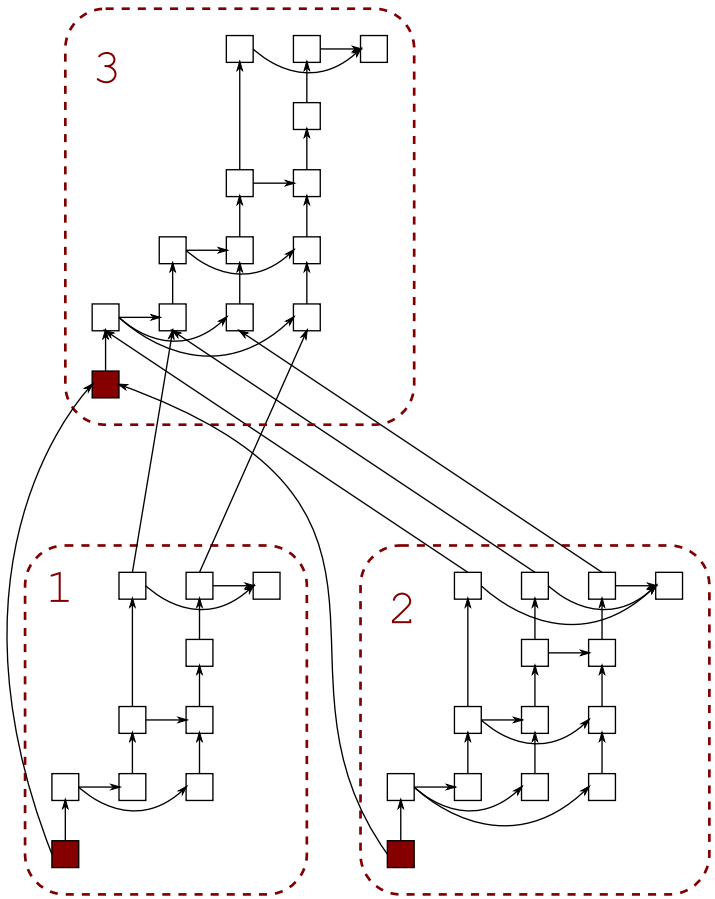


Our approach: We give to StarPU a limited view of the DAG; this is achieved by defining tasks that submit ohter tasks.

The multifrontal QR factorization: StarPU integration

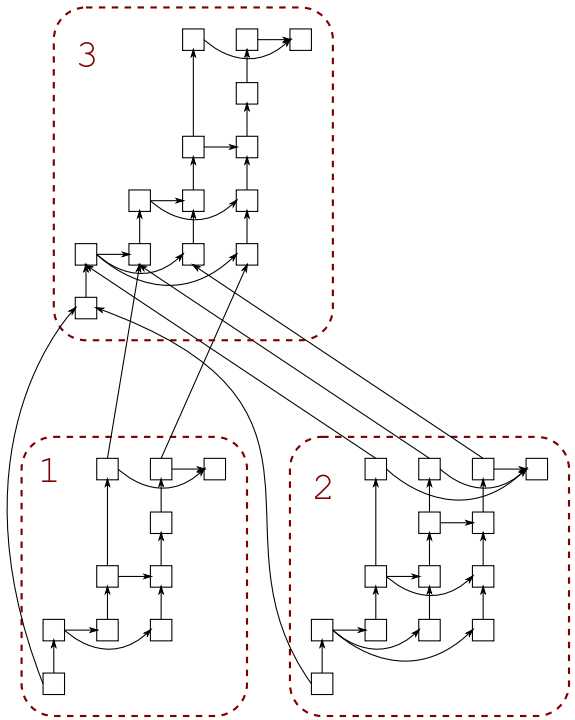
In the DAG we define

- activation tasks, i.e., tasks in charge of allocating the memory and preparing the data structures needed for processing a front



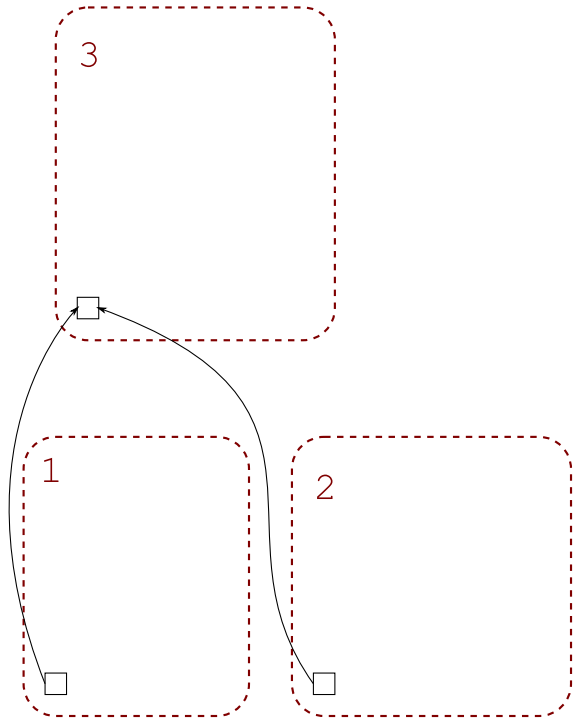
The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



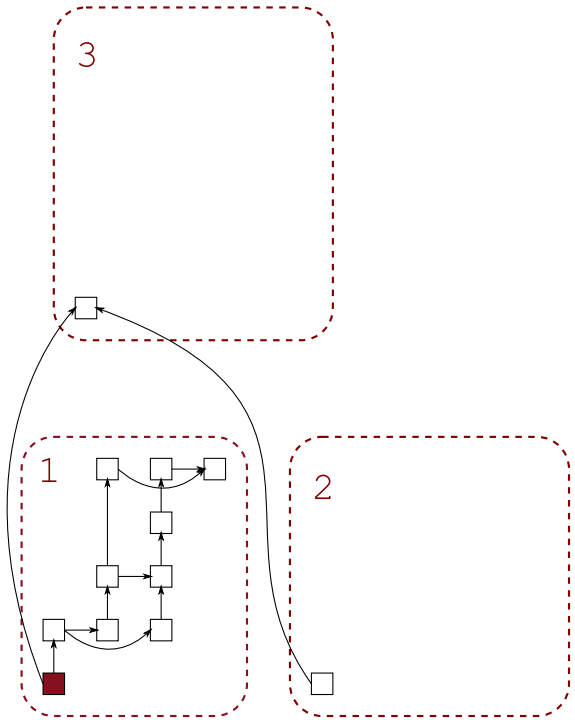
The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



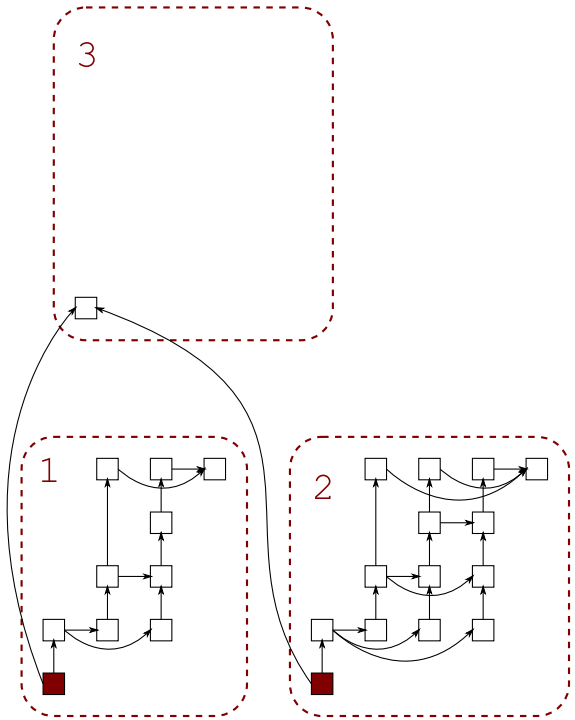
The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



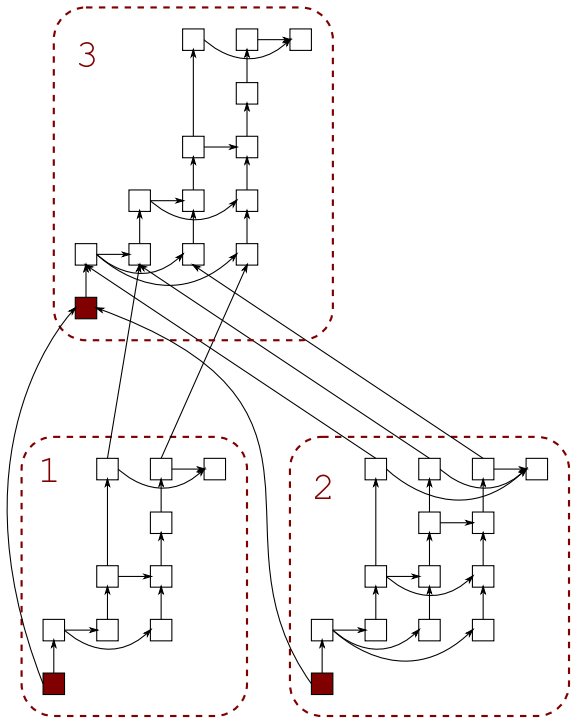
The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



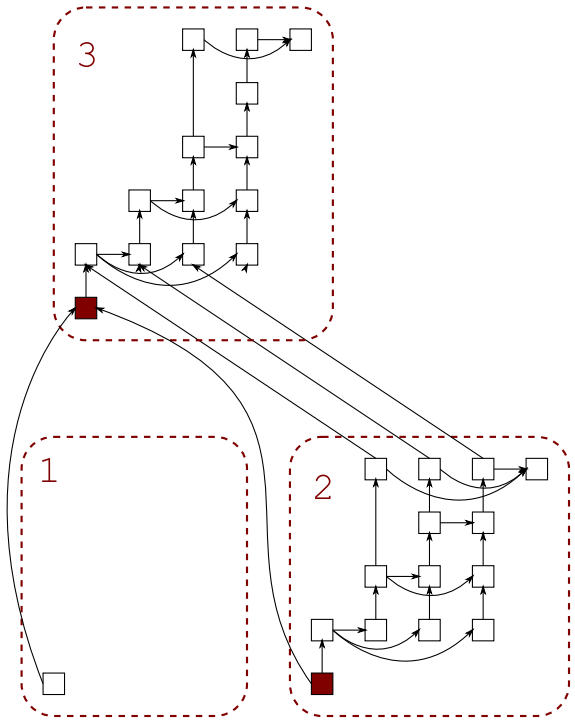
The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



The multifrontal QR factorization: StarPU integration

- All the activation tasks are submitted at once with the right dependencies and very low priority. Each of them submits other tasks with higher priority
- The runtime handles a DAG whose size is proportional only to the number of fronts that are active at a given moment
- Tree traversal orders can be identified such that the size of this dynamic DAG is as small as possible but big enough to feed all the threads



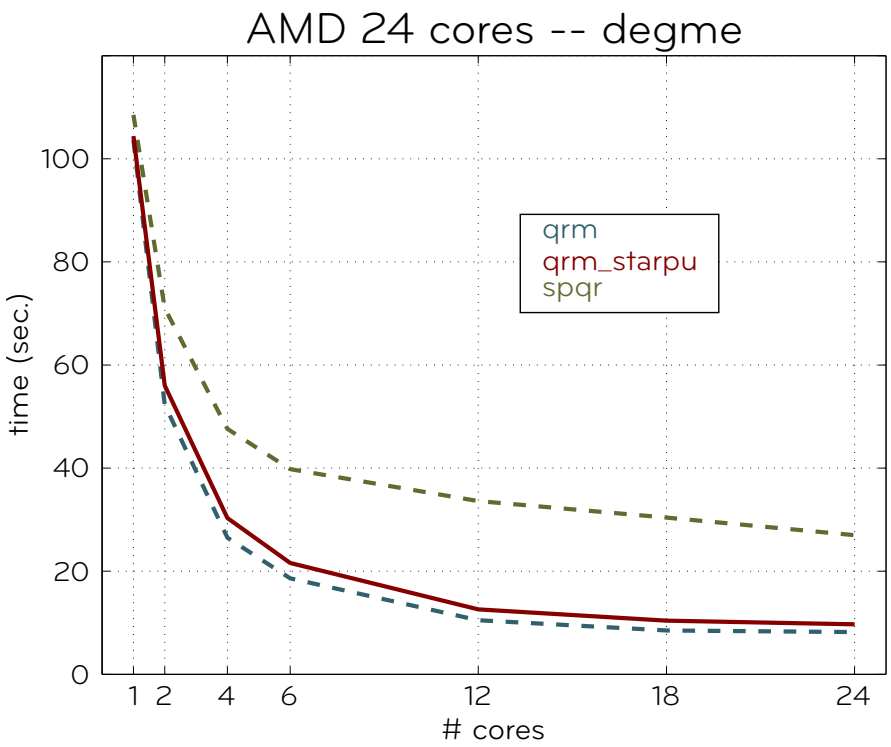
Experimental setup

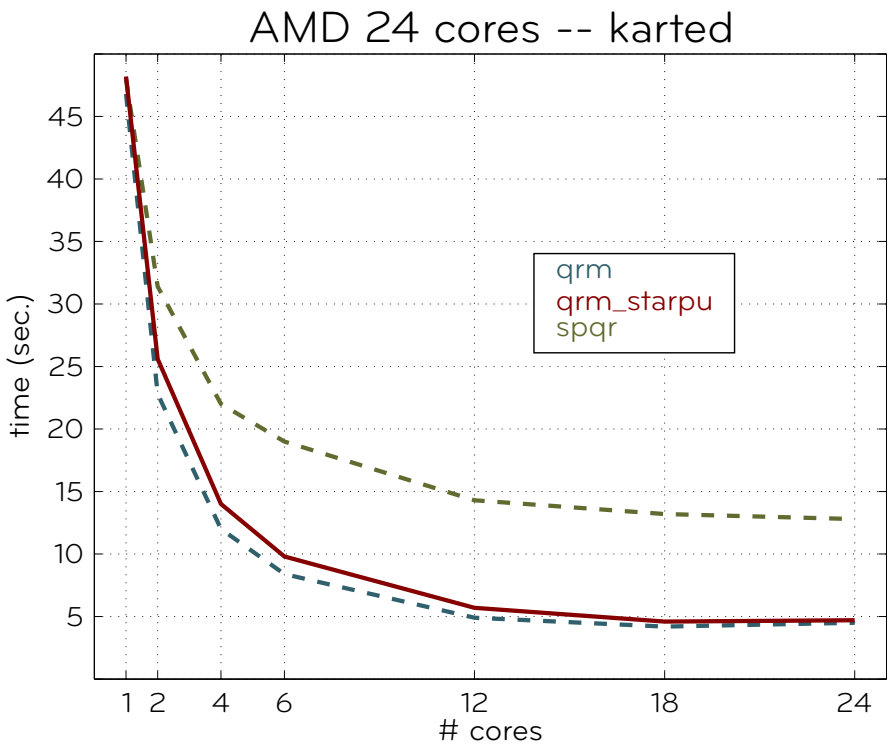
- Platform:
  - 4× AMD hexacore
  - 76 GB of memory (in 4 NUMA modules)
  - GNU 4.4 compilers
  - MKL 10.2

- Problems: some relatively small matrices from the UF collection

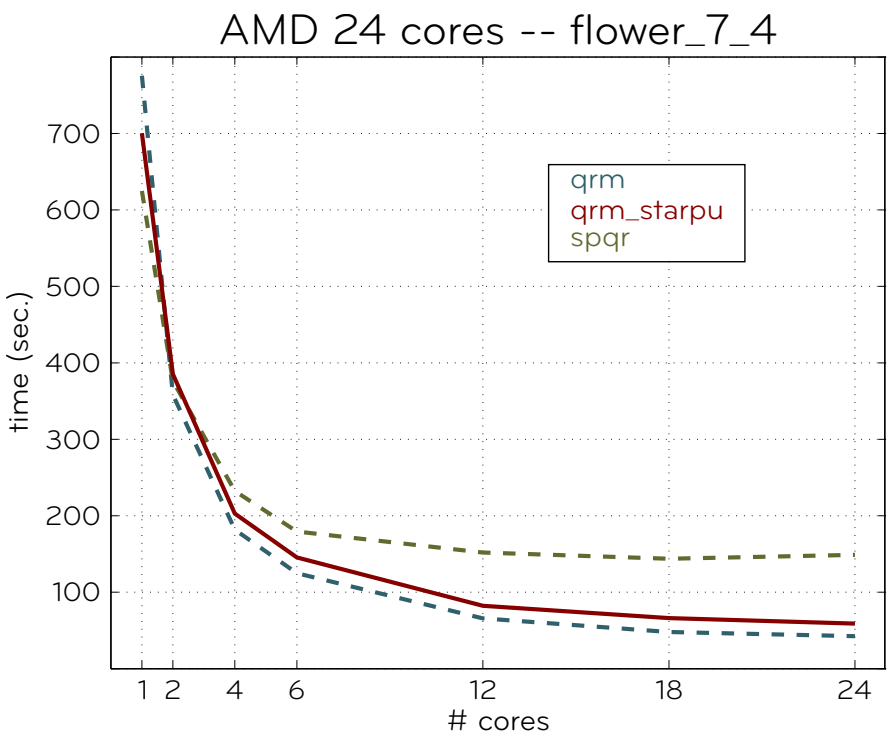
Matrix	m	n	nnz	flops
degme	185501	659415	8127528	591 G
karted	46502	133115	1770349	258 G
flower_7_4	27693	67593	202218	4261 G
EternityII_E	11077	262144	1503732	544 G
cat_ears_4_4	19020	44448	132888	716 G
tp-6	142752	1014301	11537419	255 G

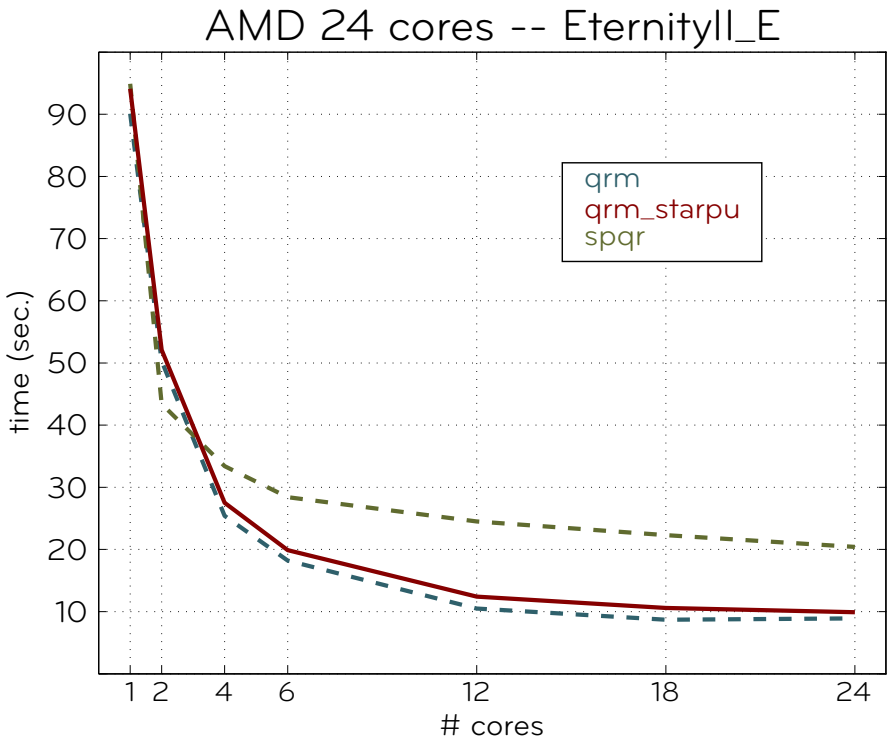
Experimental results



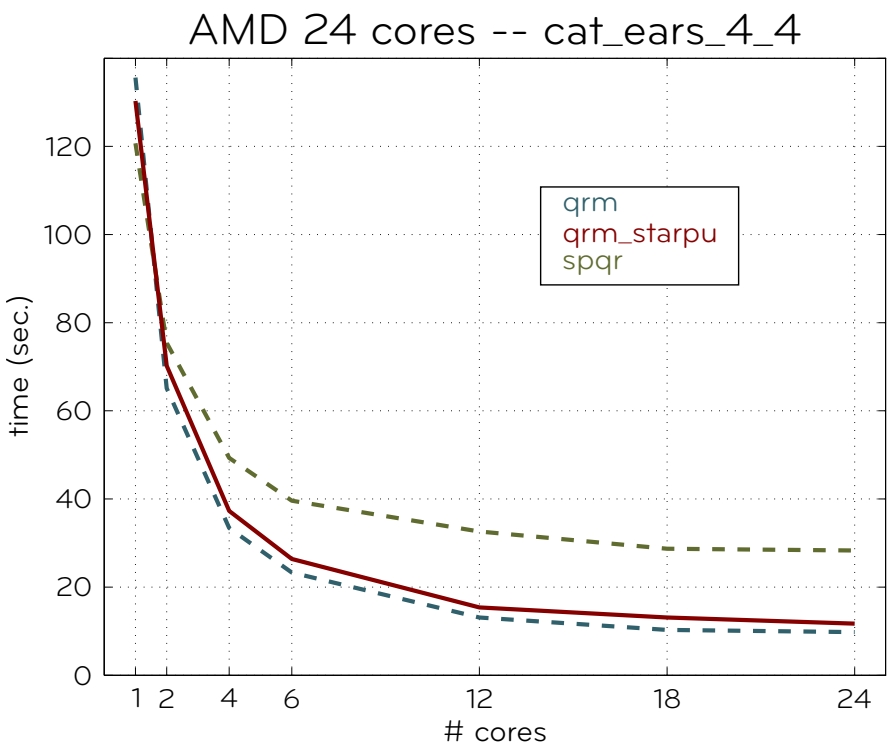


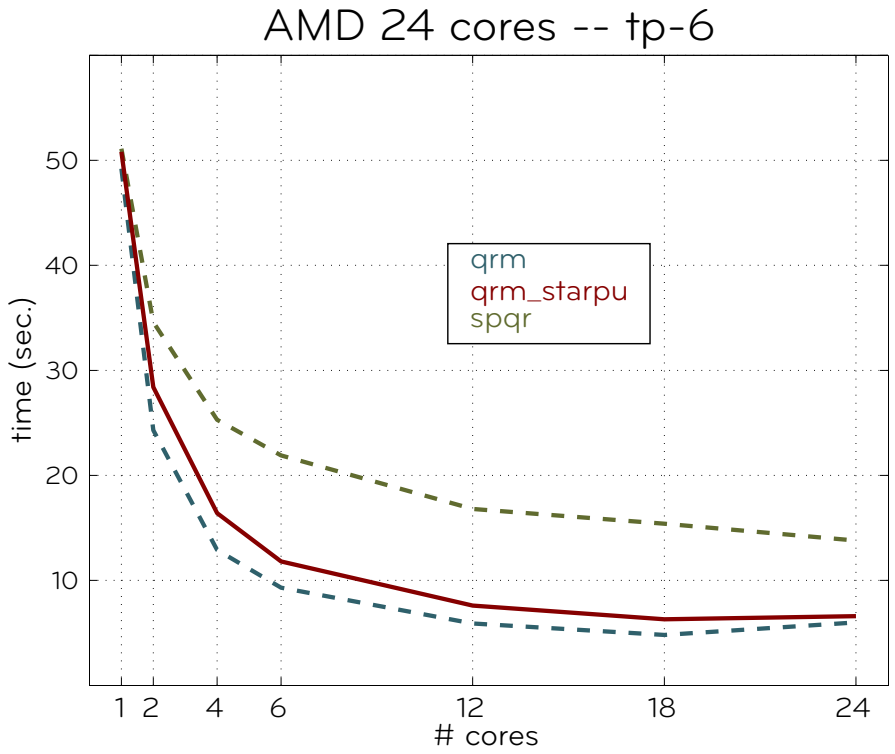
Experimental results





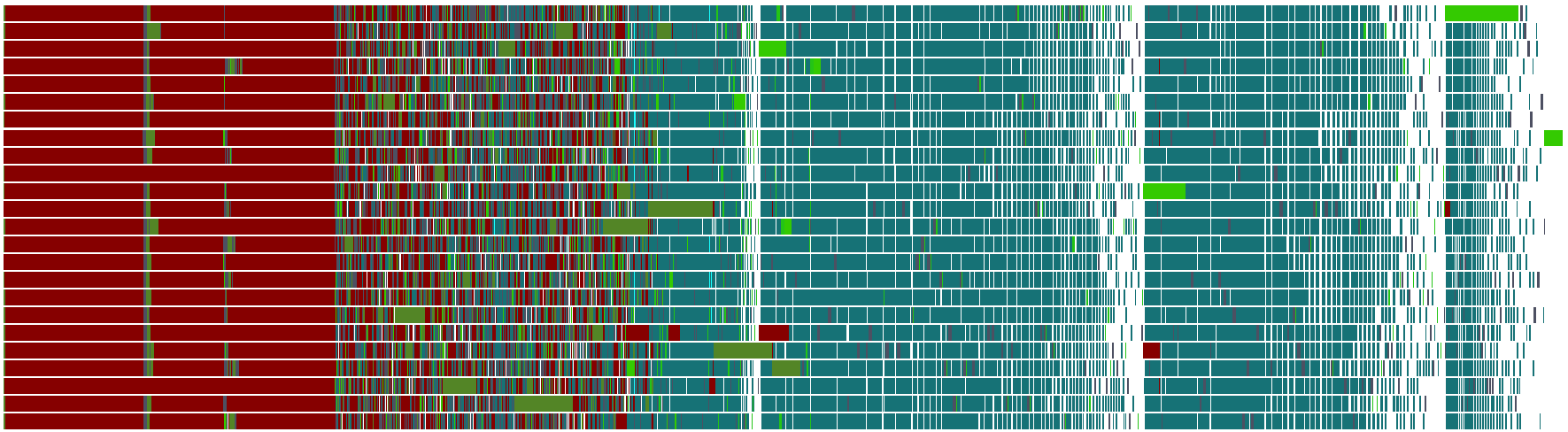
Experimental results





Experimental results

Execution trace for the `degme` matrix:



Two main problems can be identified:

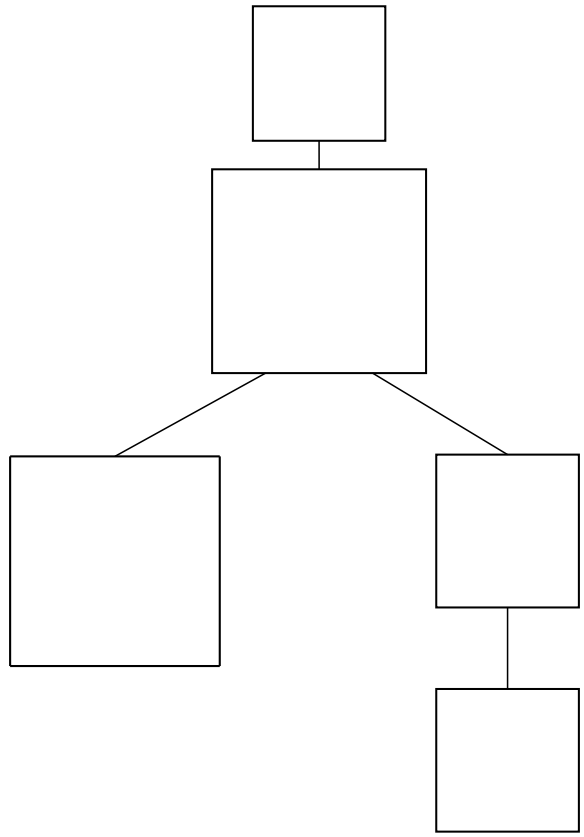
- Too much time is spent into tasks submission (in red). The issue is under investigation
- At the moment, parent-child dependencies are not finely managed which means that it is not possible to start working on a node until all of its children are completely factorized

Multifrontal Cholesky: front factorization on CPU-GPU hybrid systems

From elimination tree to frontal matrix

Bottom-up traversal of the elimination tree.
At each vertex (front):

- Assembling of contribution blocks from children
- Partial factorization of the frontal matrix

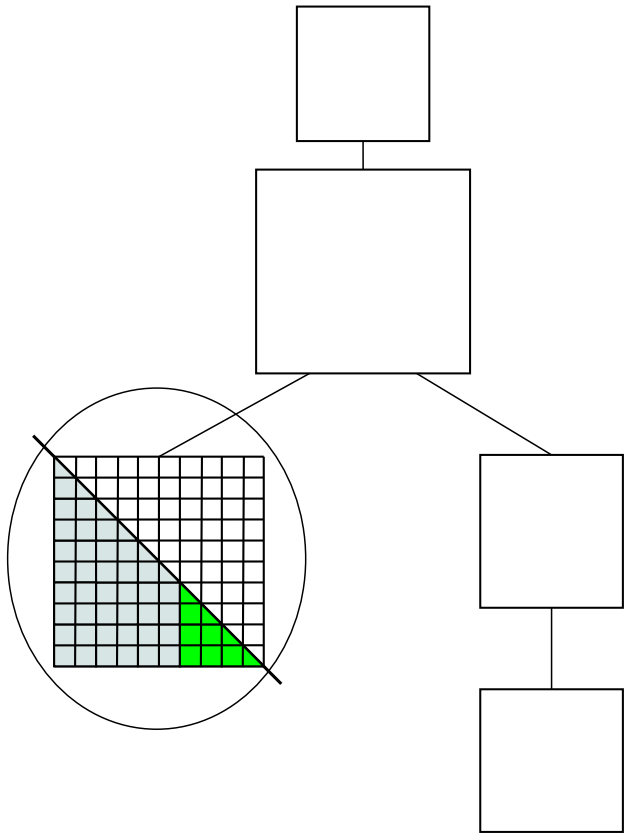


Elimination tree

From elimination tree to frontal matrix

Bottom-up traversal of the elimination tree.
At each vertex (front):

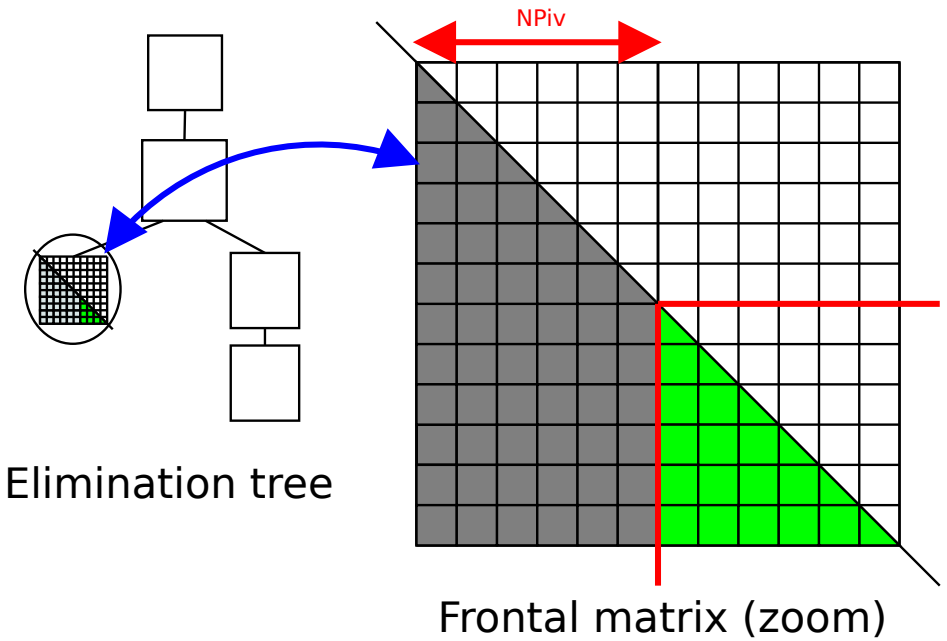
- Assembling of contribution blocks from children
- Partial factorization of the frontal matrix



Elimination tree

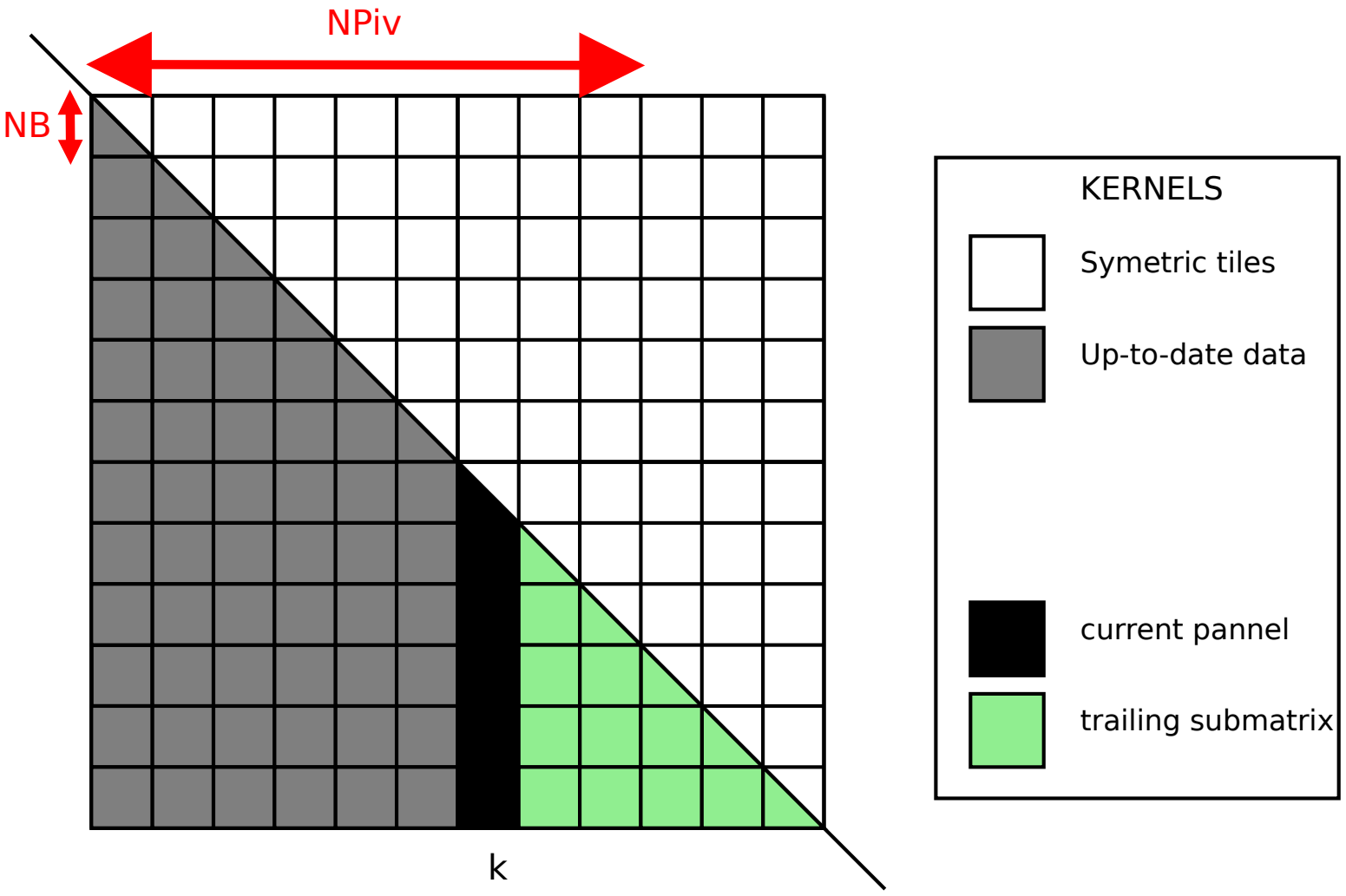
Tile Cholesky front factorization

- Derived from: *A class of parallel tiled linear algebra algorithms for multicore architectures*. Buttari et al., Parallel Comput., 2009
- Extension: Partial factorization of NPiv variables and computation of the Schur complement
- Use of a runtime system: StarPU



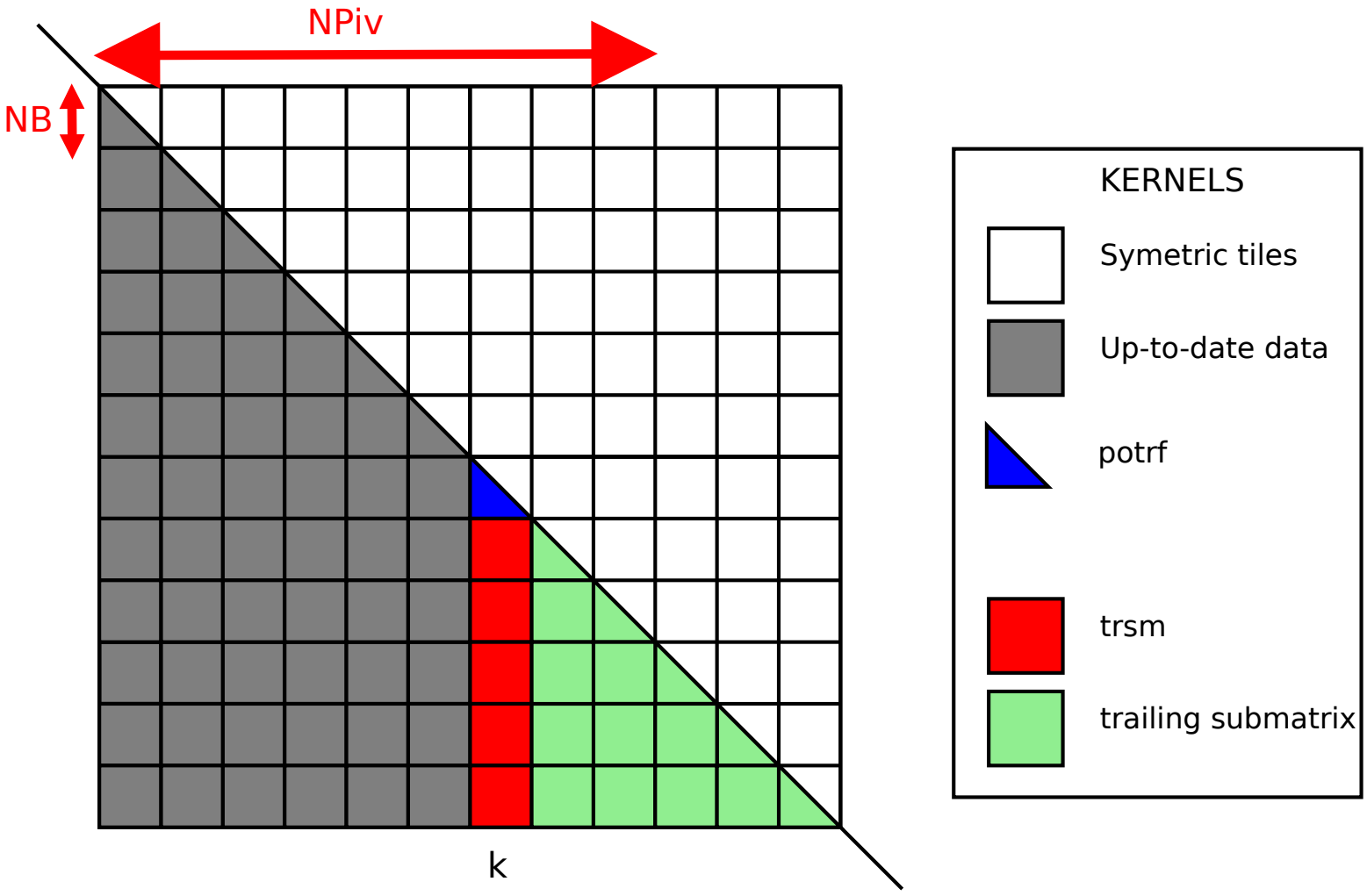
Tile Cholesky front factorization, *NB*|*NPiv* case

- NB: tile size
- NPiv: number of pivots



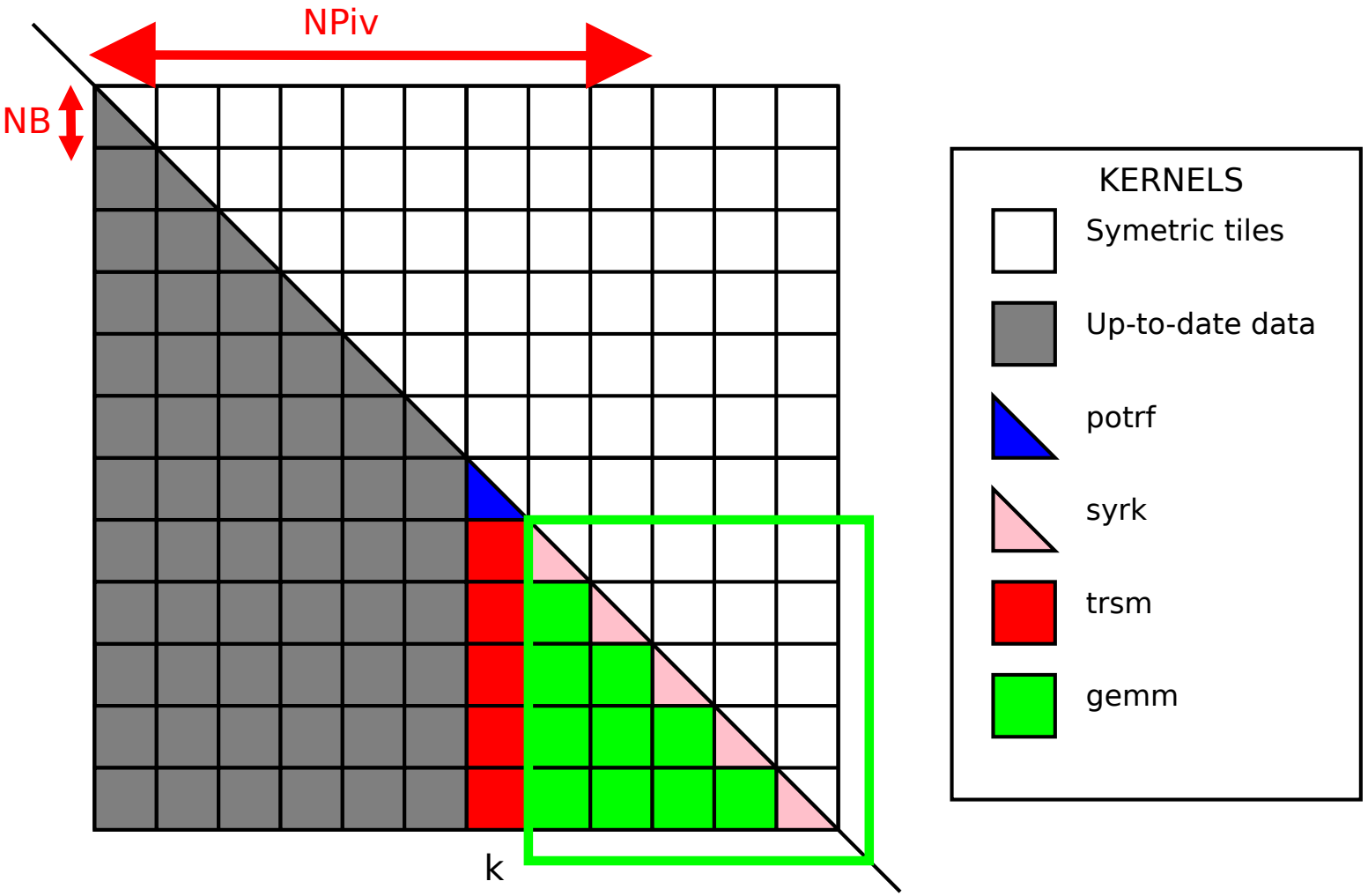
Tile Cholesky front factorization, *NB*|*NPiv* case

- NB: tile size
- NPiv: number of pivots



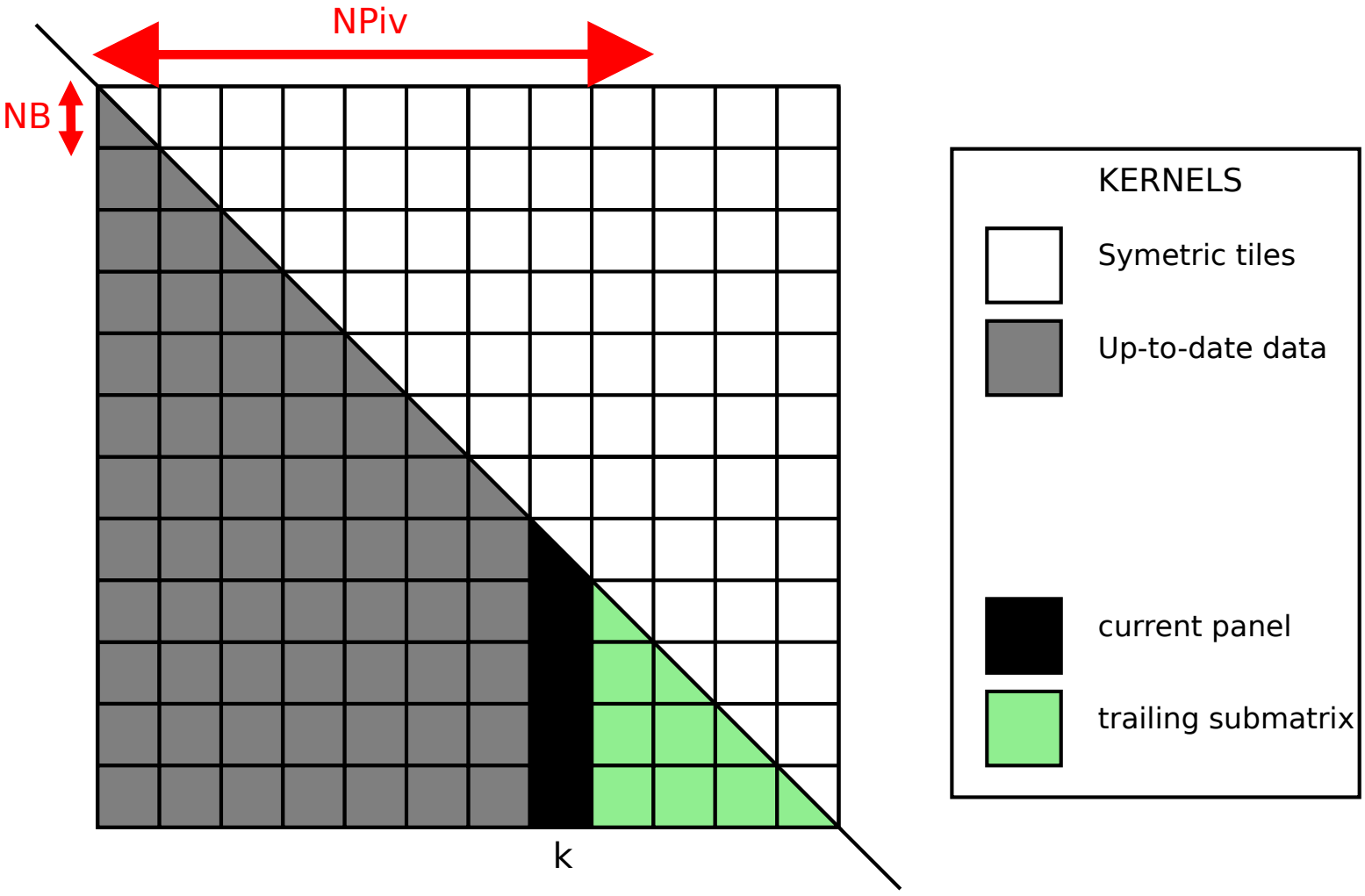
Tile Cholesky front factorization, *NB*|*NPiv* case

- NB: tile size
- NPiv: number of pivots



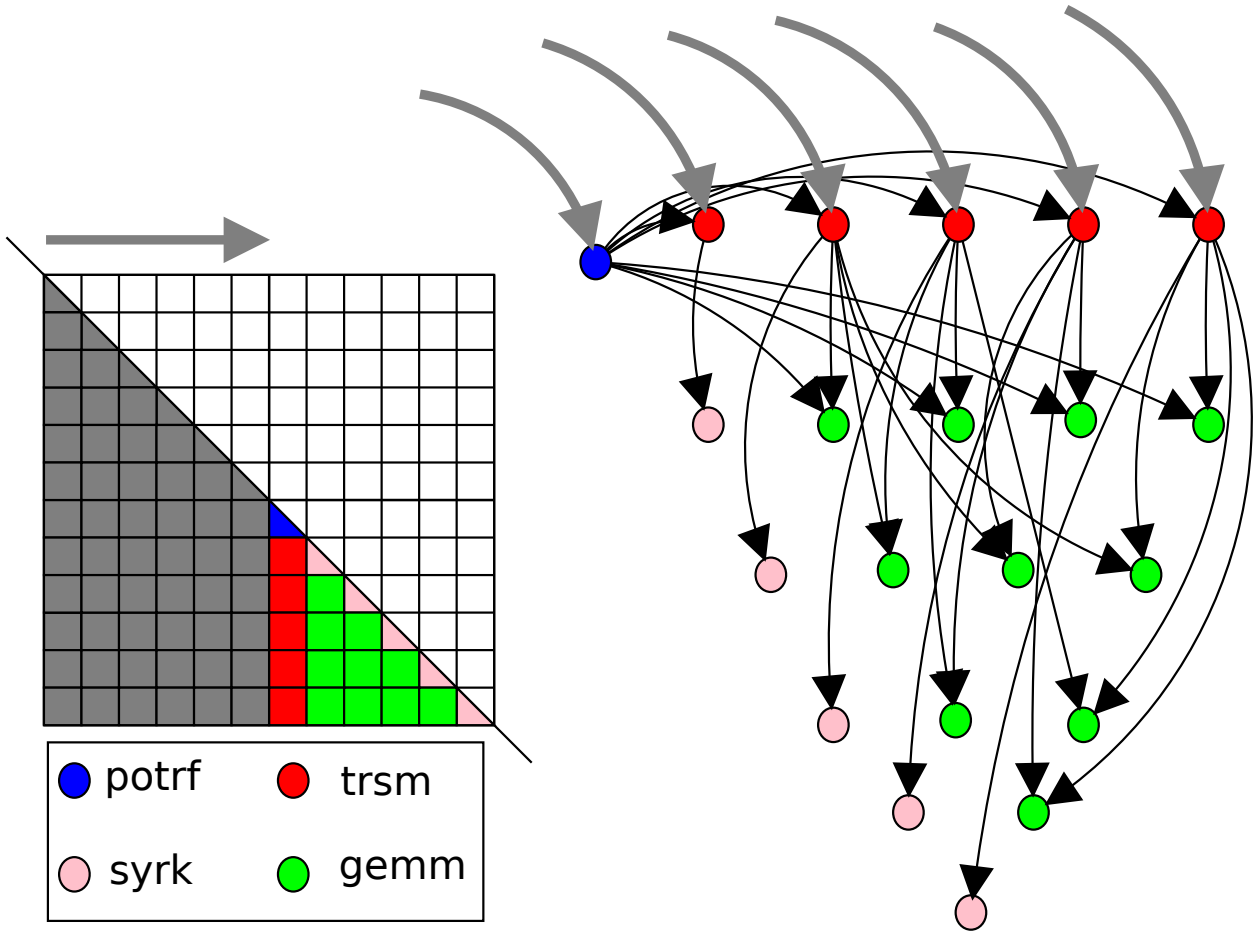
Tile Cholesky front factorization, *NB*|*NPiv* case

- NB: tile size
- NPiv: number of pivots



Task flow

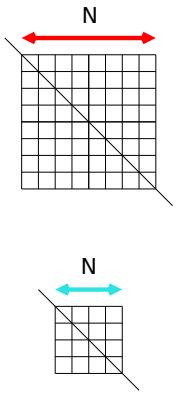
- Fine granularity
- High concurrency
- Out-of-order execution

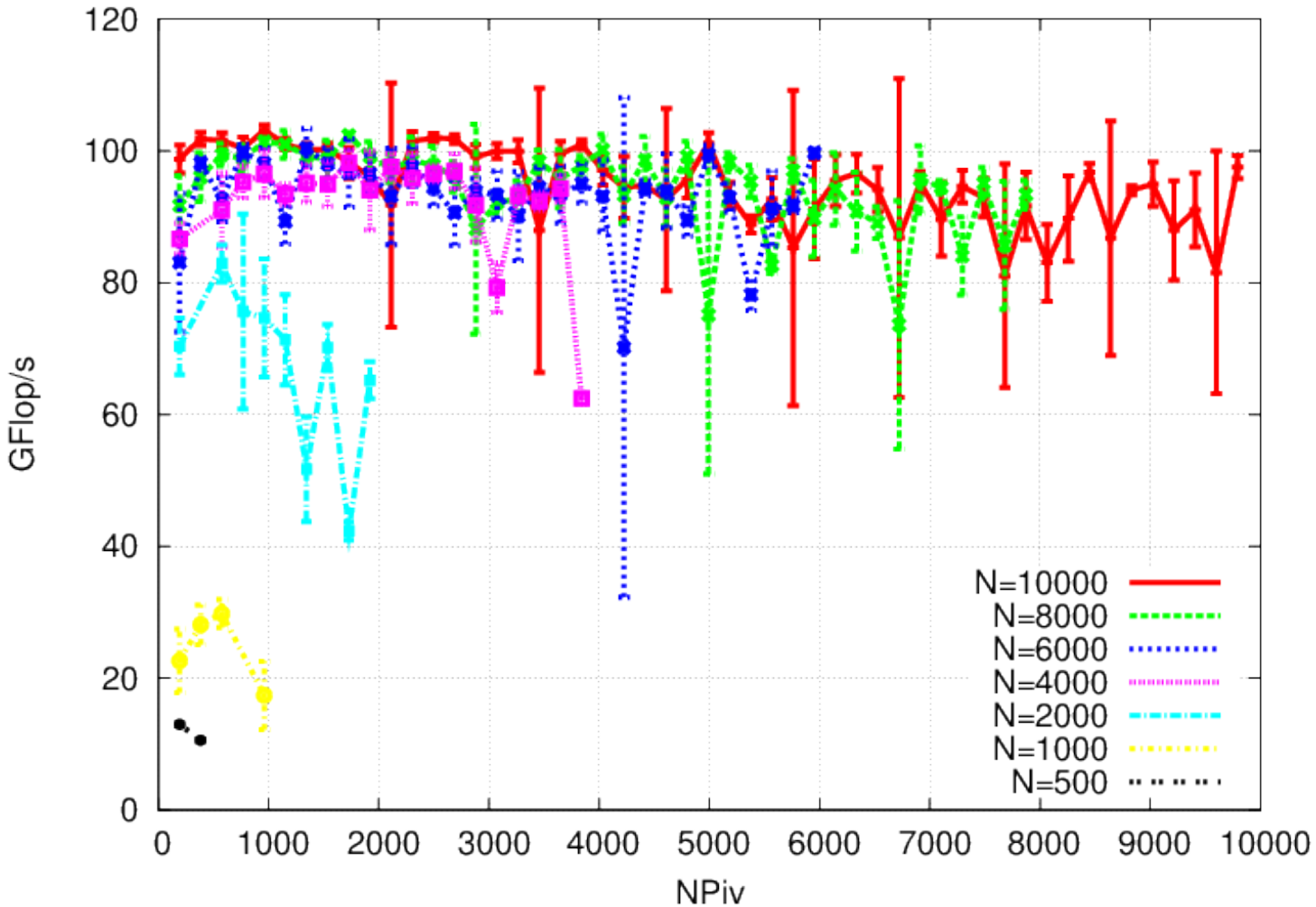


Multicore architecture

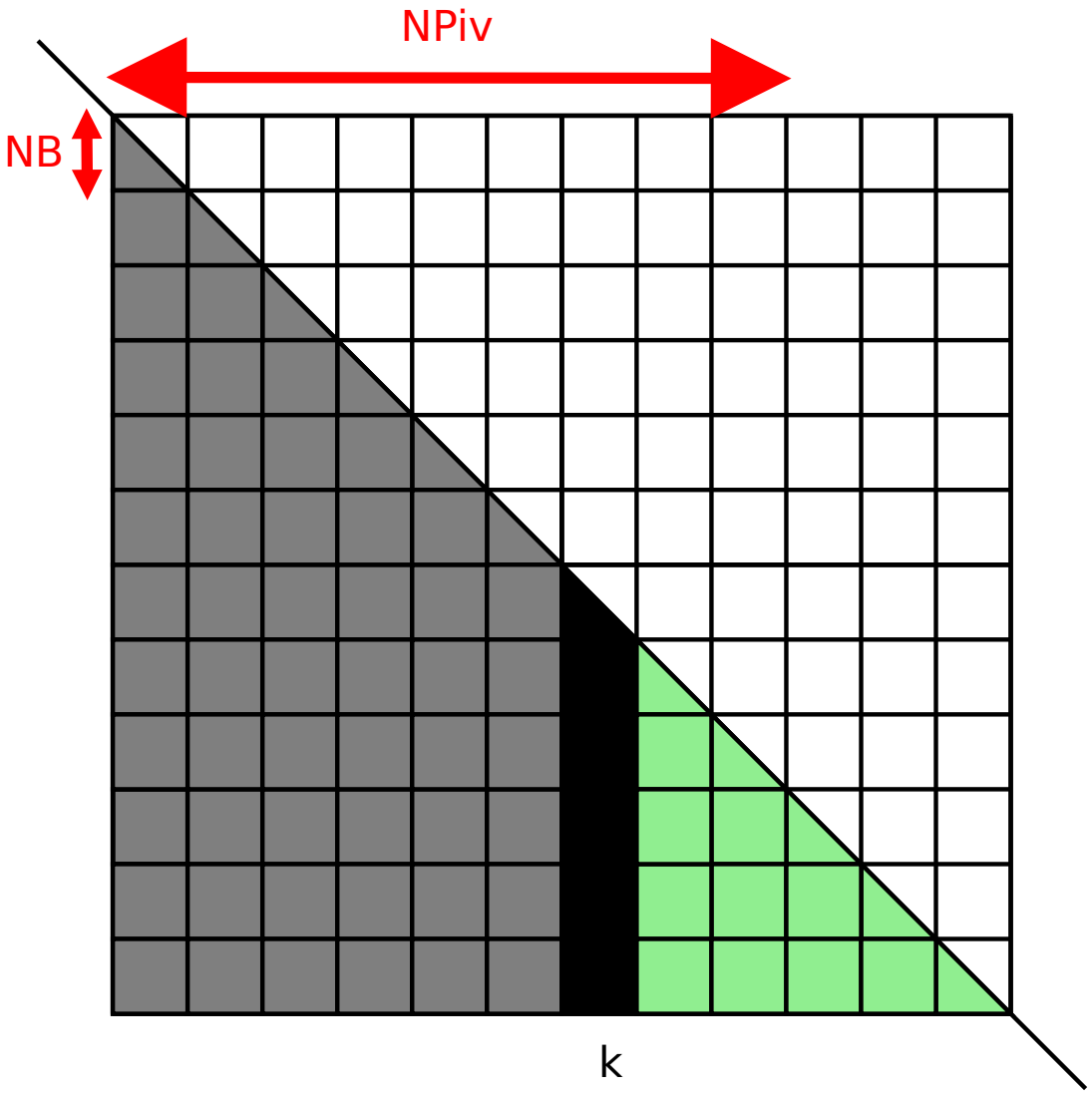
- 12 cores, 2,67 GHz Dual-socket Hexa-core Westmere Intel Xeon X5650 processor
- 12 MB L3 cache
- 36 GB RAM

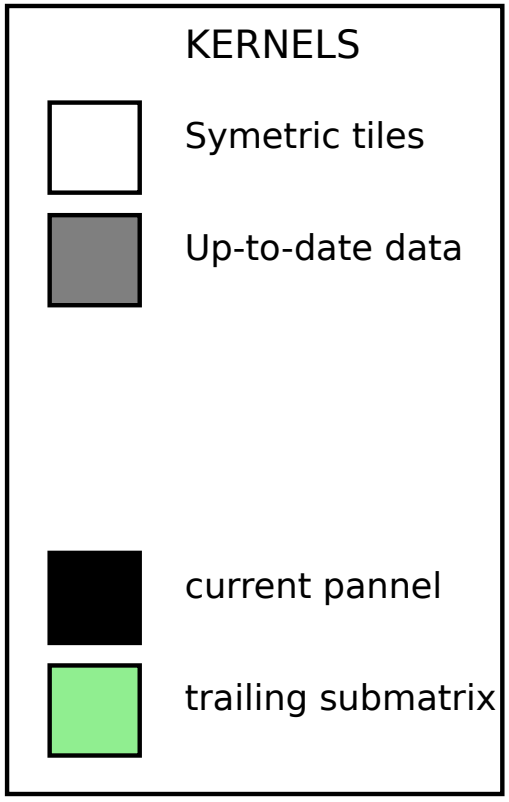
Performance (multicore architecture)



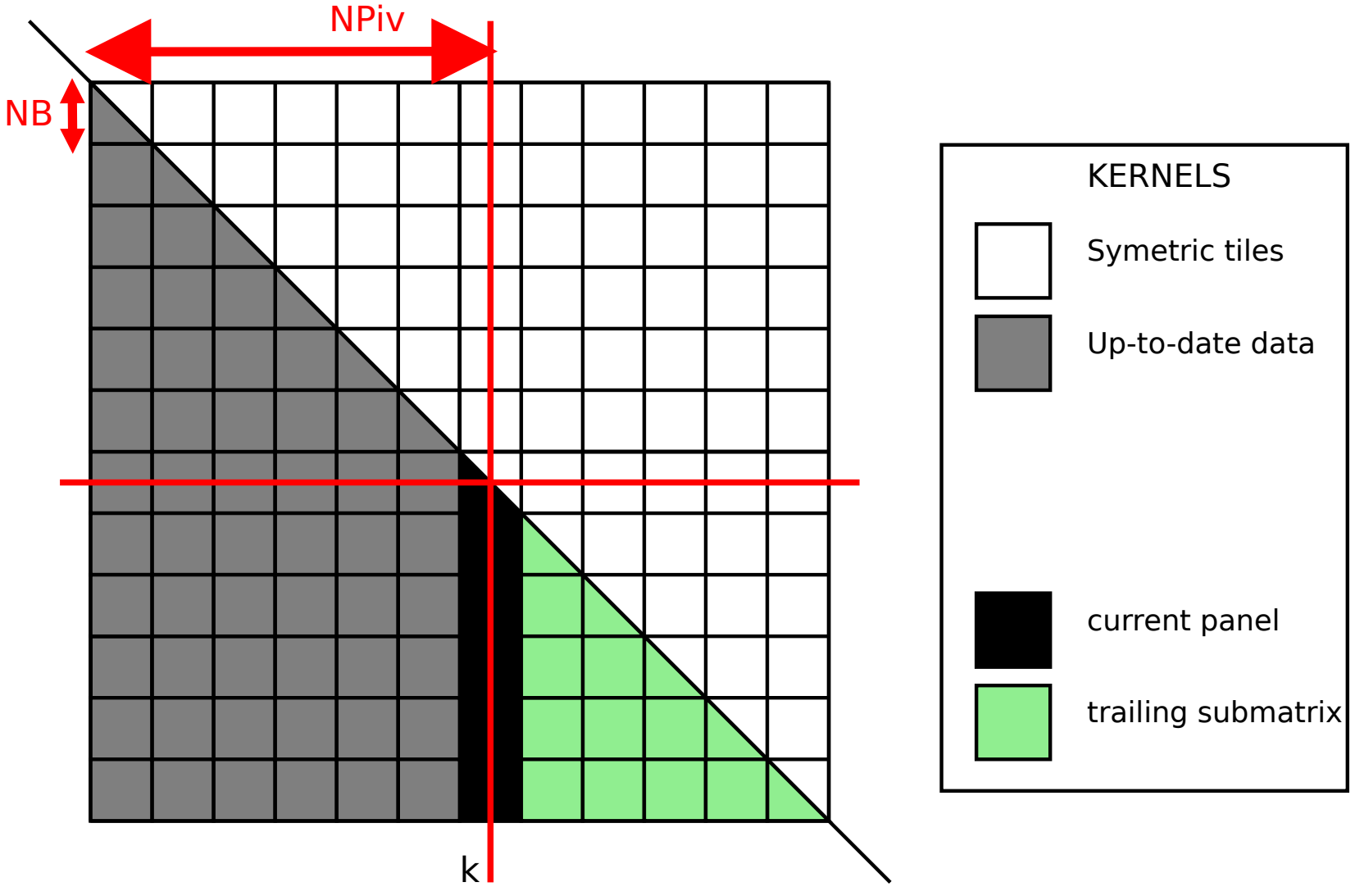


Cholesky front factorization, general case (any NPiv)

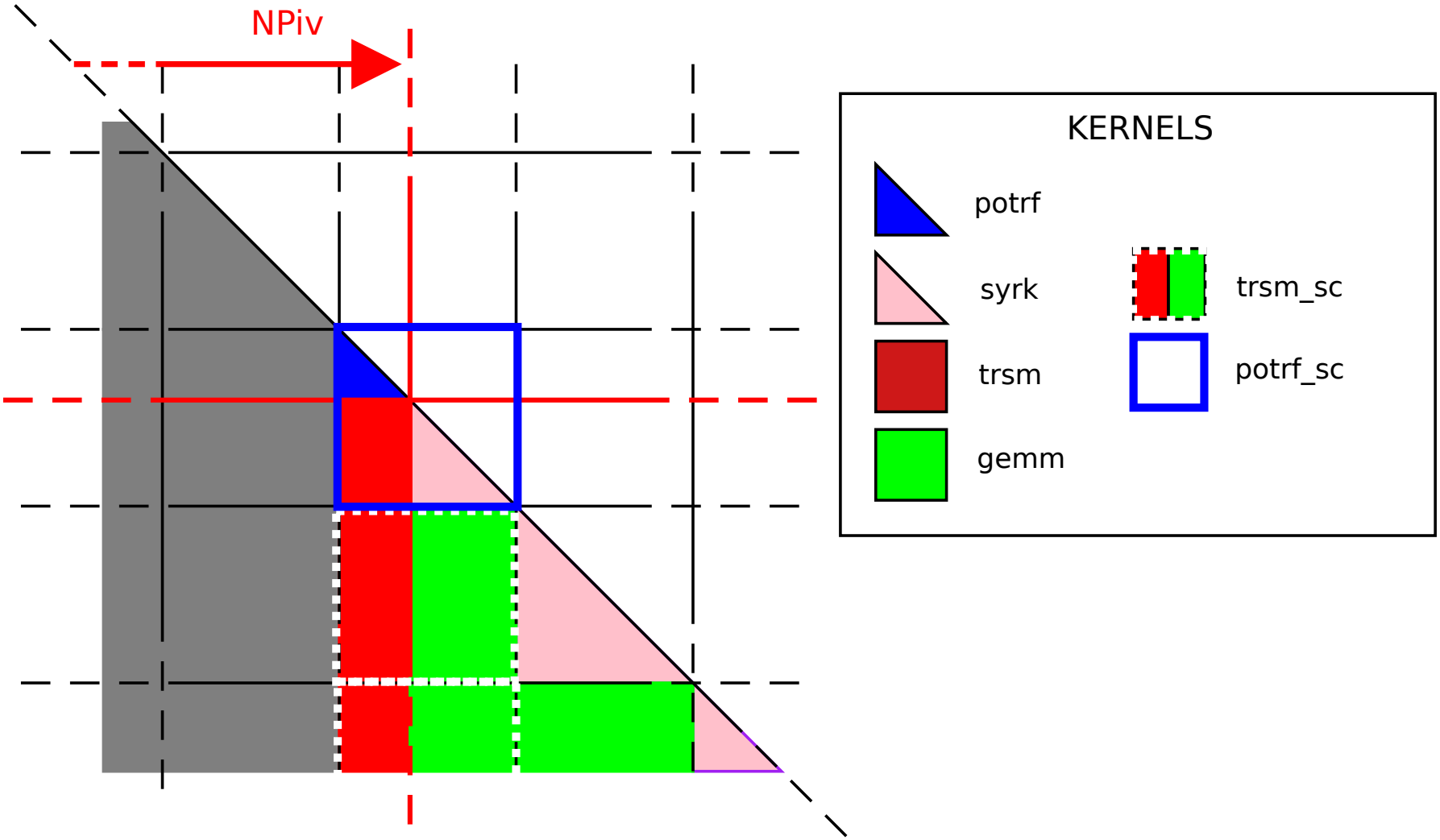




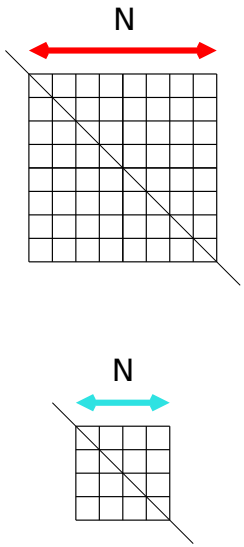
Cholesky front factorization, general case (any NPiv)

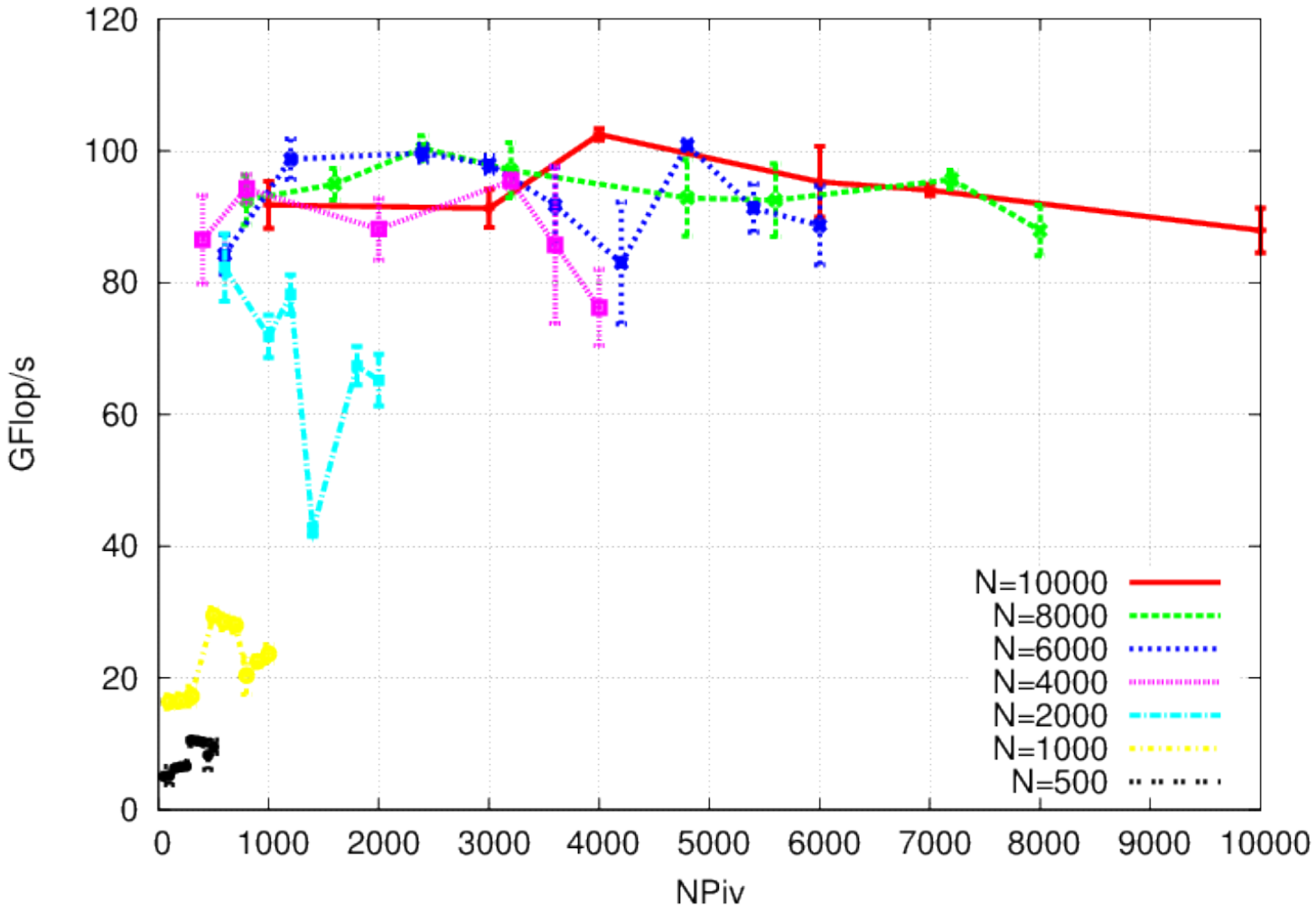


Cholesky front factorization, general case (any NPiv)



Performance (multicore architecture)

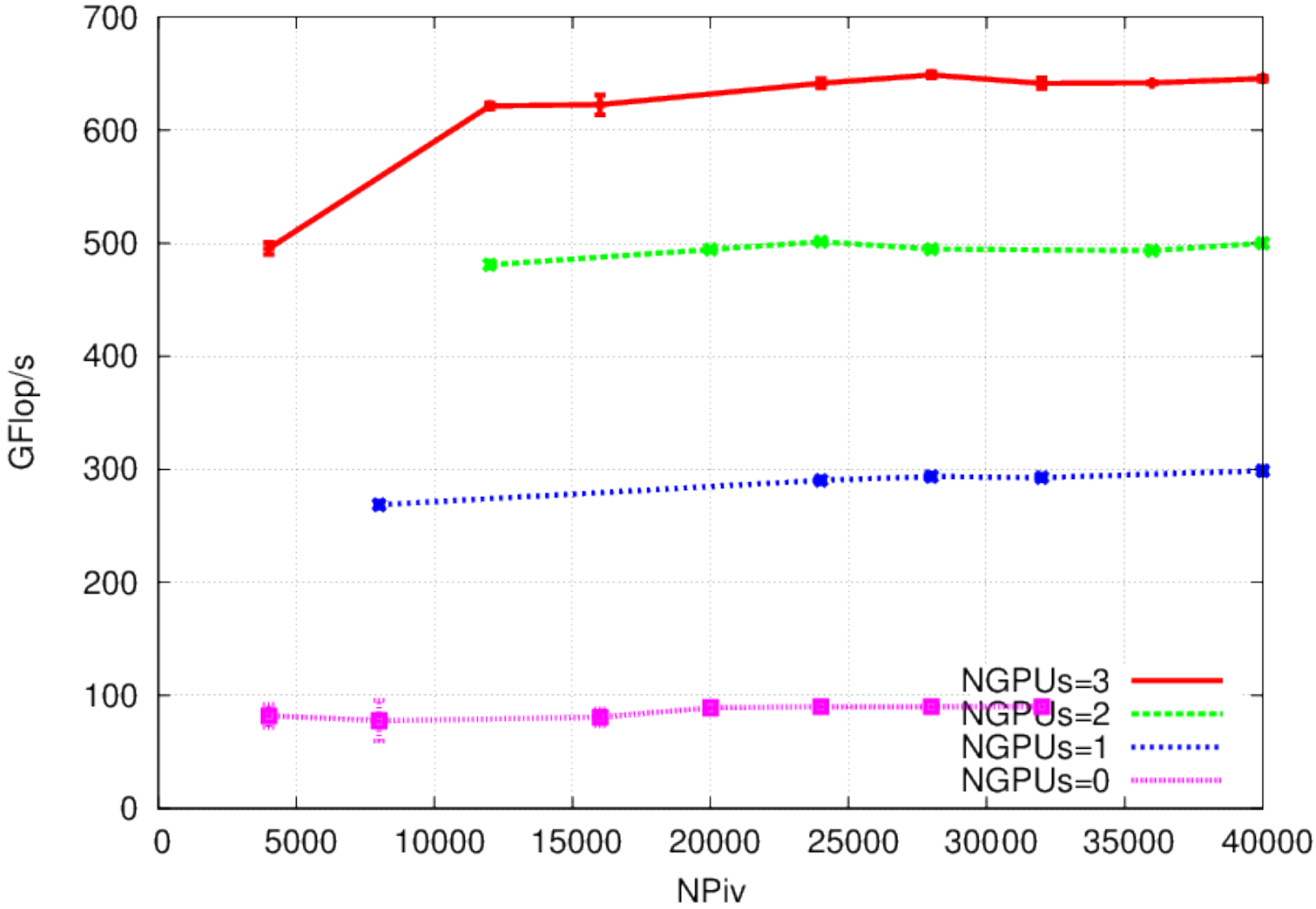




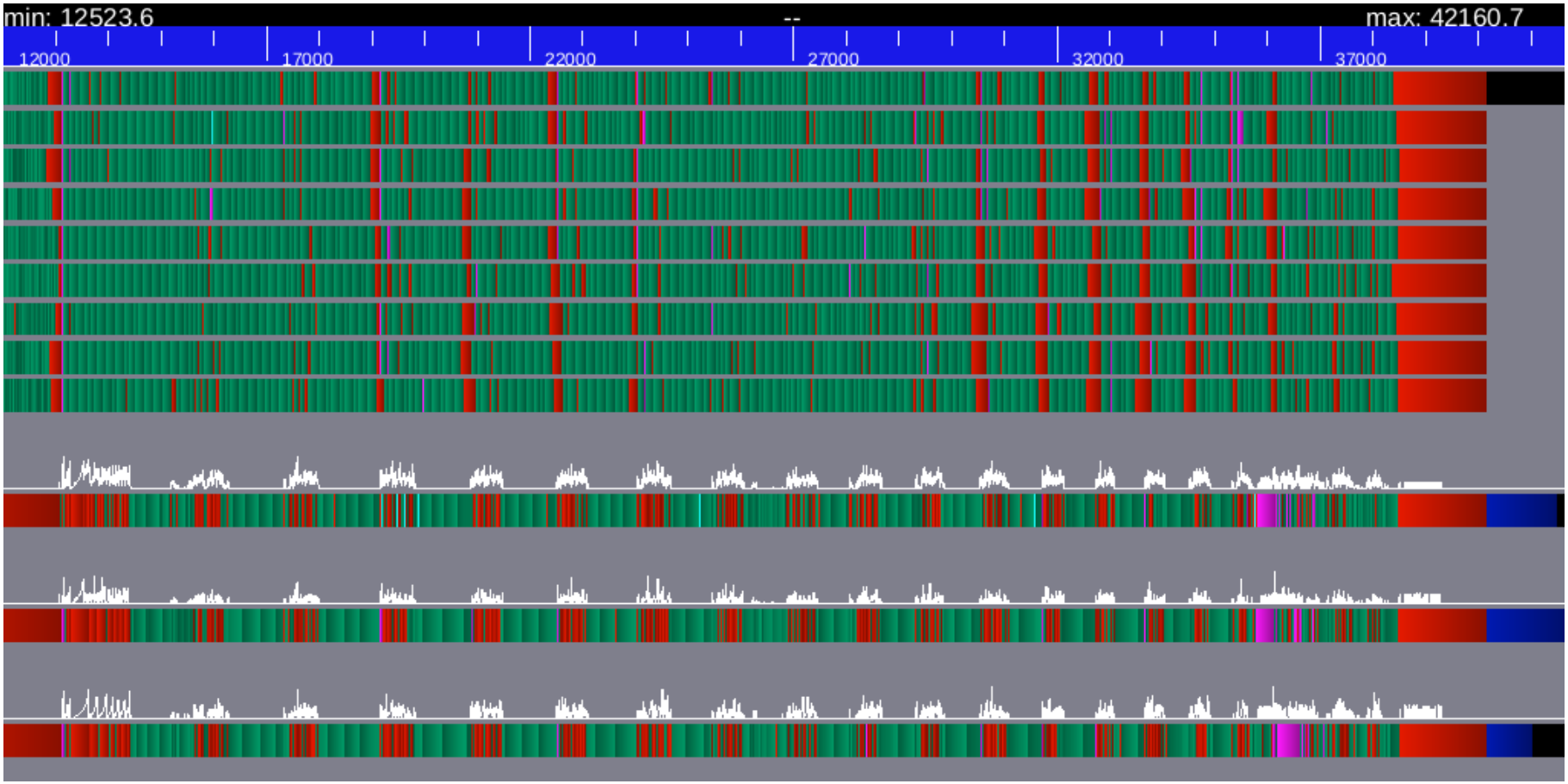
CPU-GPU hybrid architecture

- 12 cores, 2,67 GHz Dual-socket Hexa-core Westmere Intel Xeon X5650 processor
- 12 MB L3 cache
- 36 GB RAM
- 3 NVIDIA Tesla M2070 (Fermi) GPUs

Performance (CPU-GPU hybrid architecture) - N=40K



Execution trace (CPU-GPU hybrid architecture)



Conclusion, perspectives

Conclusion

- Preliminary work towards a functioning multifrontal code
- Not as efficient (for now) as codes designed for a specific architecture
- Performance portability across architectures
- Dense kernels to be released in the MAGMA library

Future work

- Robust solver (Cholesky assembly step, solution steps)
- Memory consumption (progressive task activation)
- Cluster of heterogeneous nodes
- Investigate explicit data dependencies management (DAGuE)

Future work

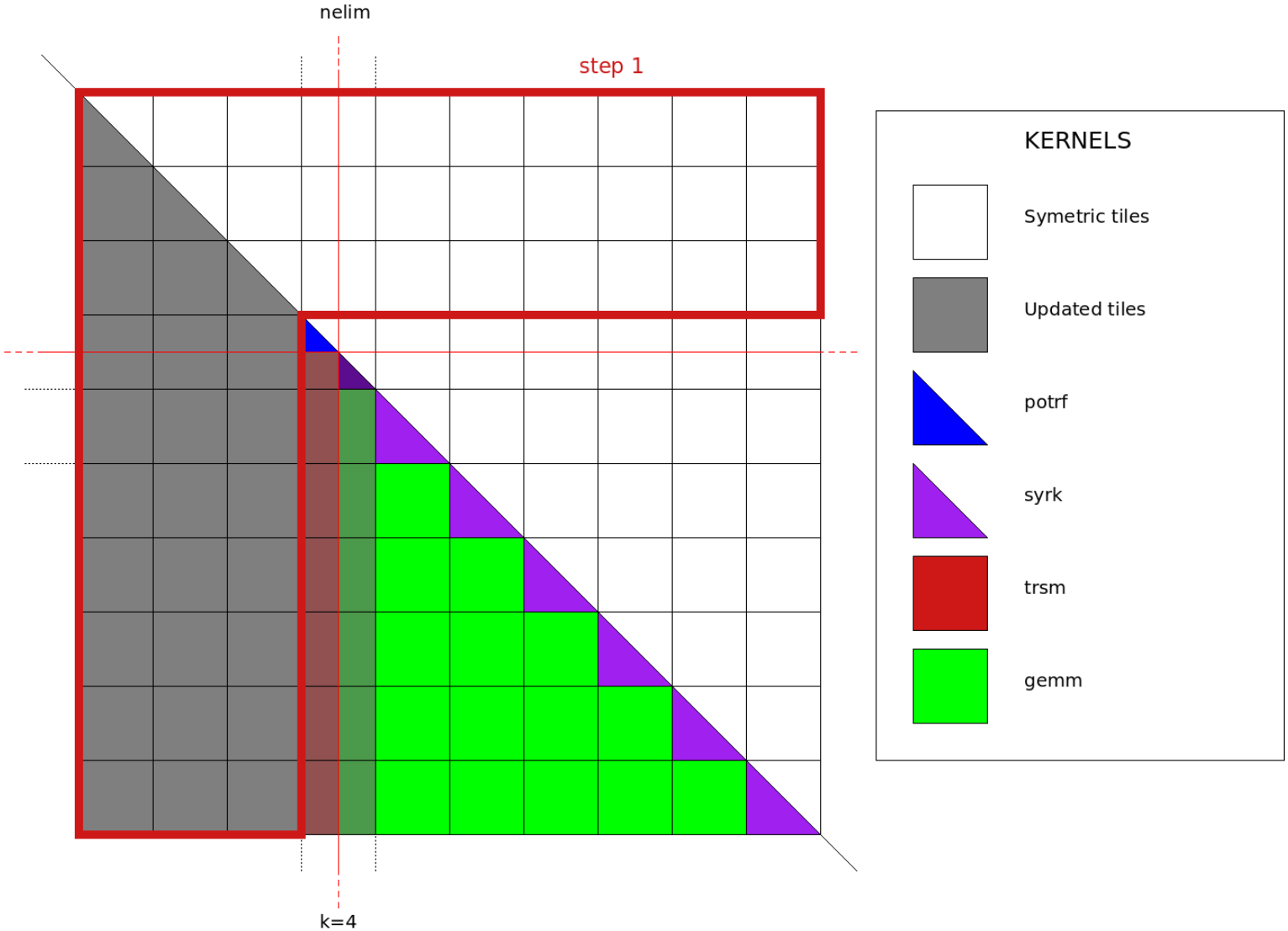
- Robust solver (Cholesky assembly step, solution steps)
- Memory consumption (progressive task activation)
- Cluster of heterogeneous nodes
- Investigate explicit data dependencies management (DAGuE)

Thanks!

Appendix

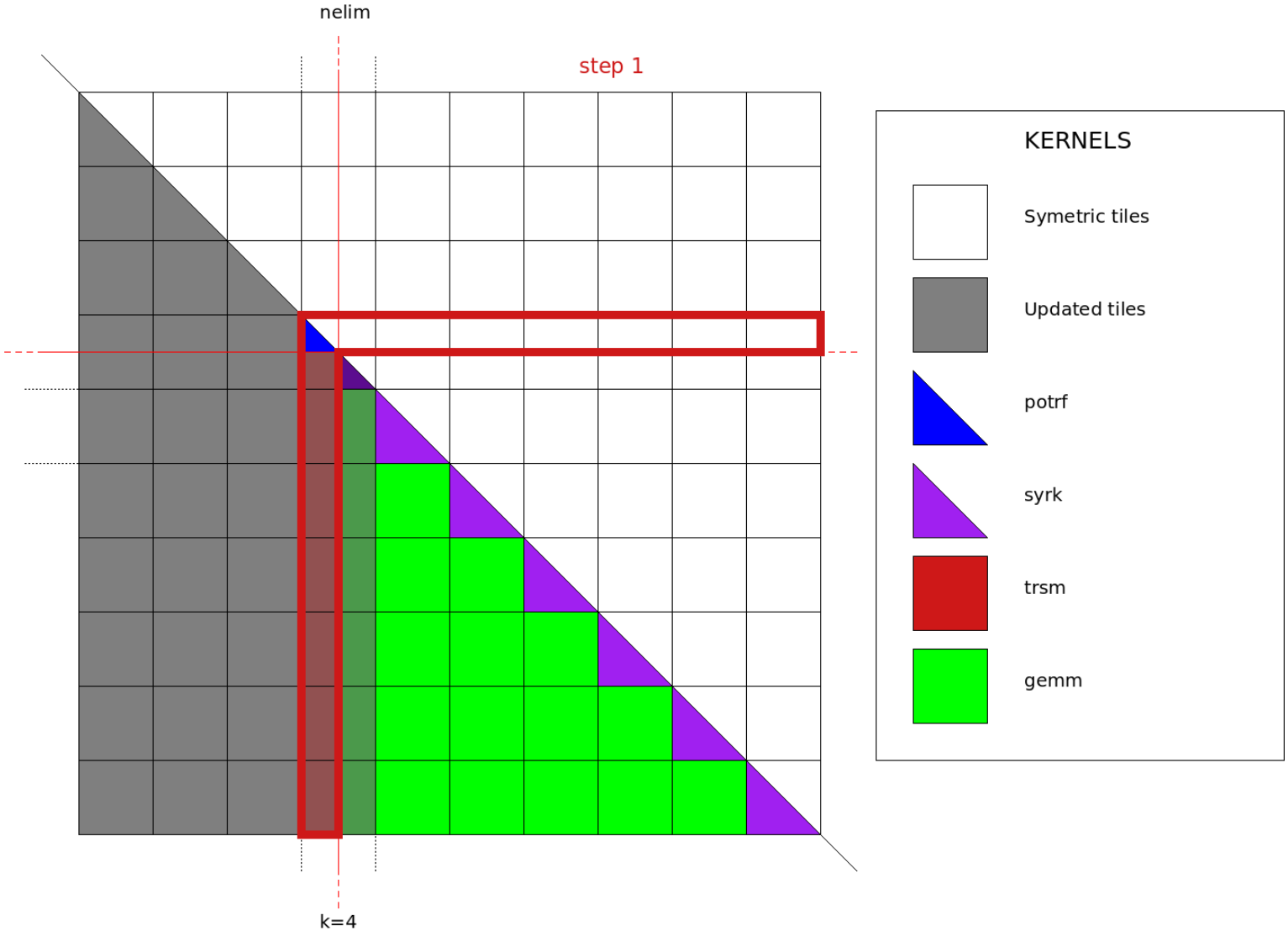
Validation (step 1)

- Factorization on nelim



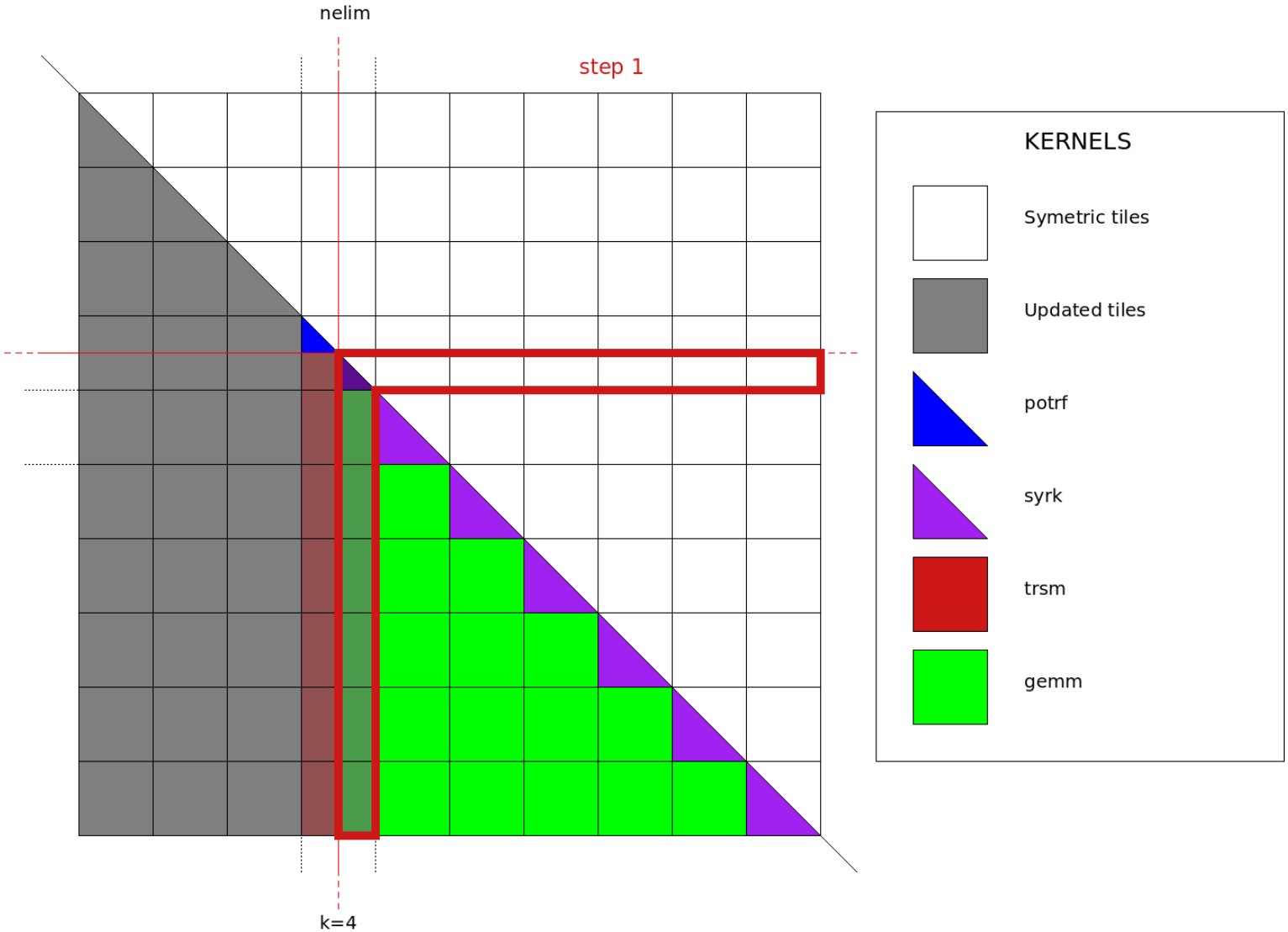
Validation (step 1)

- Factorization on nelim



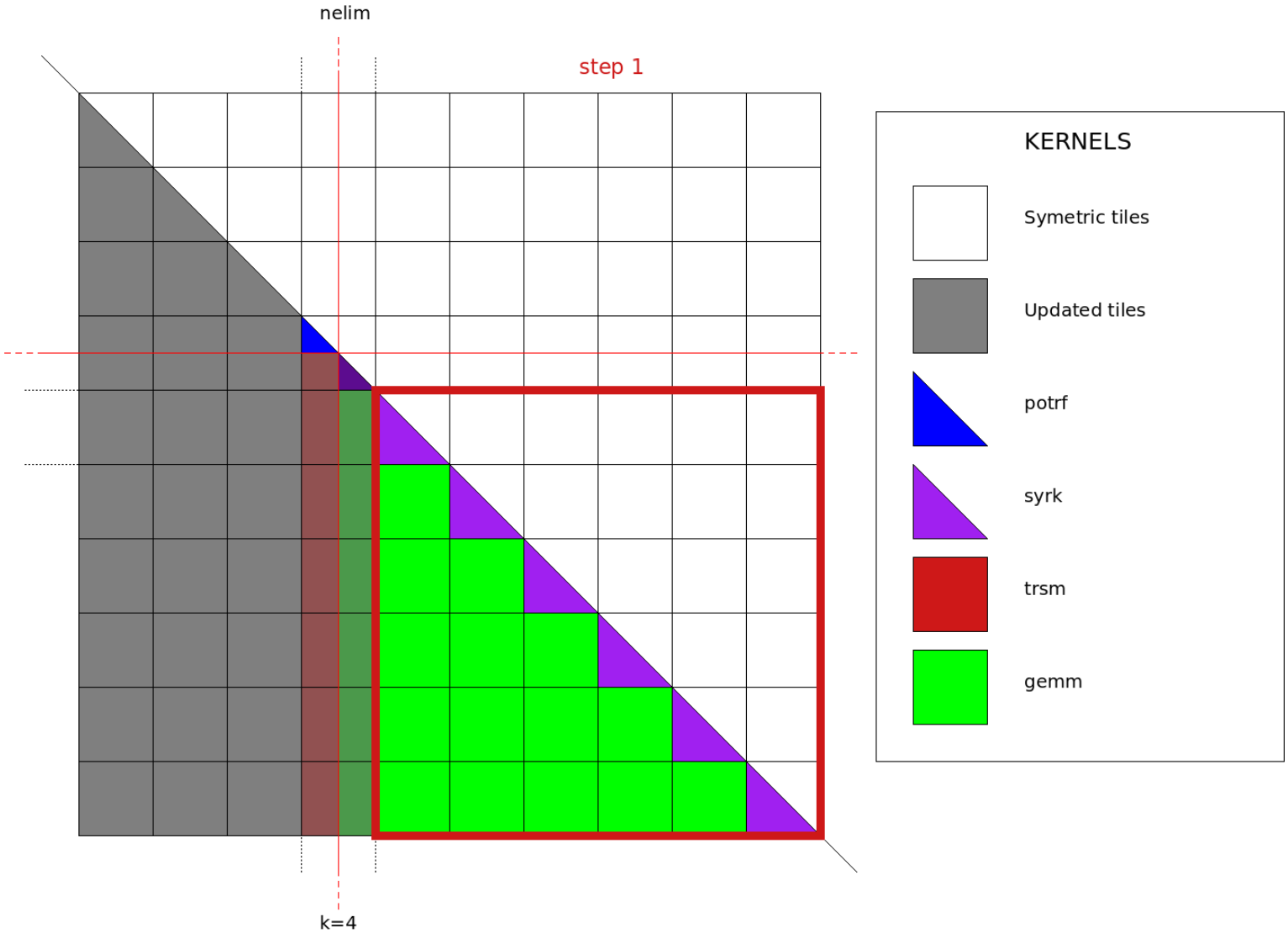
Validation (step 1)

- Update the trailing submatrix



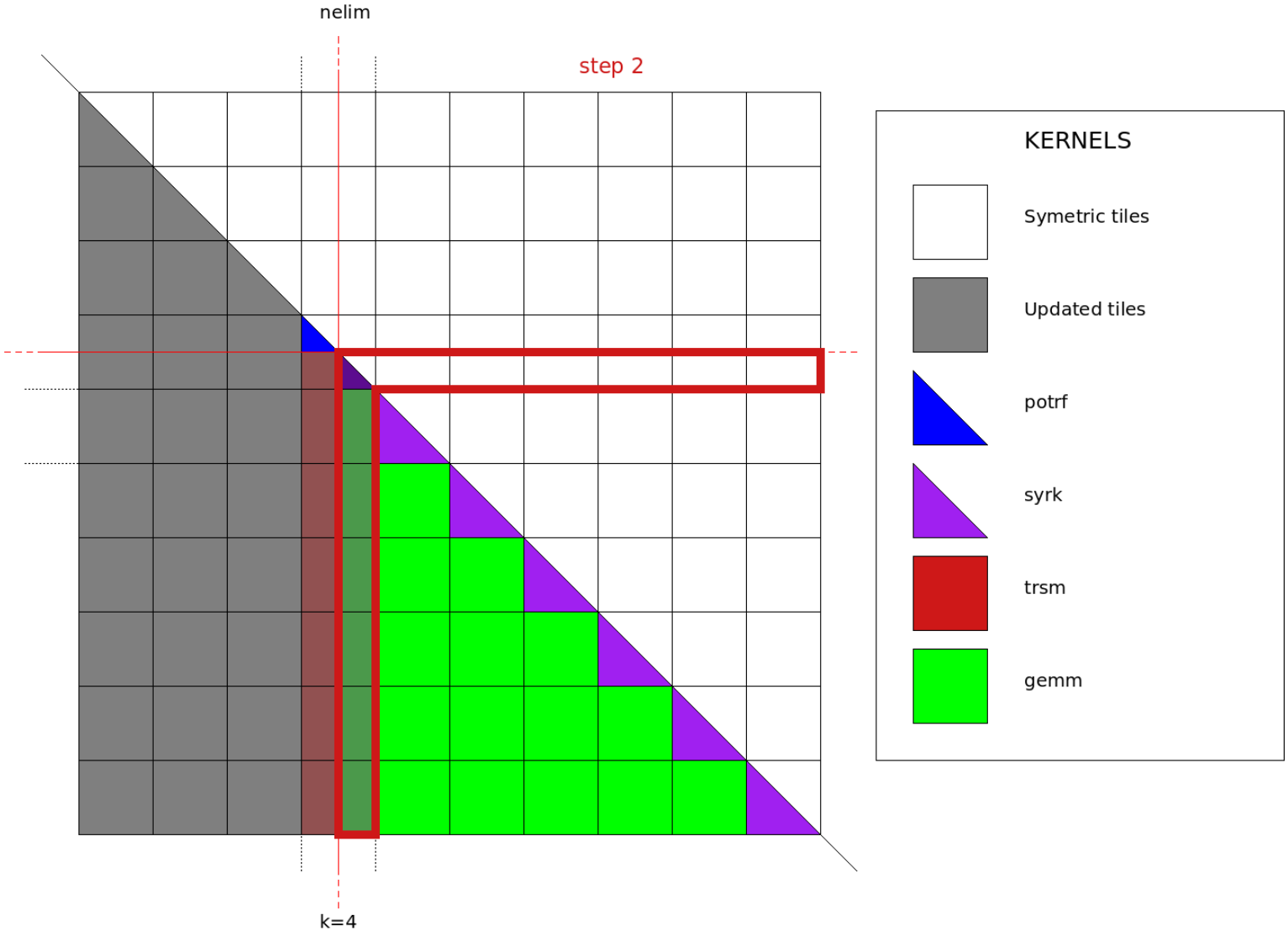
Validation (step 1)

- Update the trailing submatrix



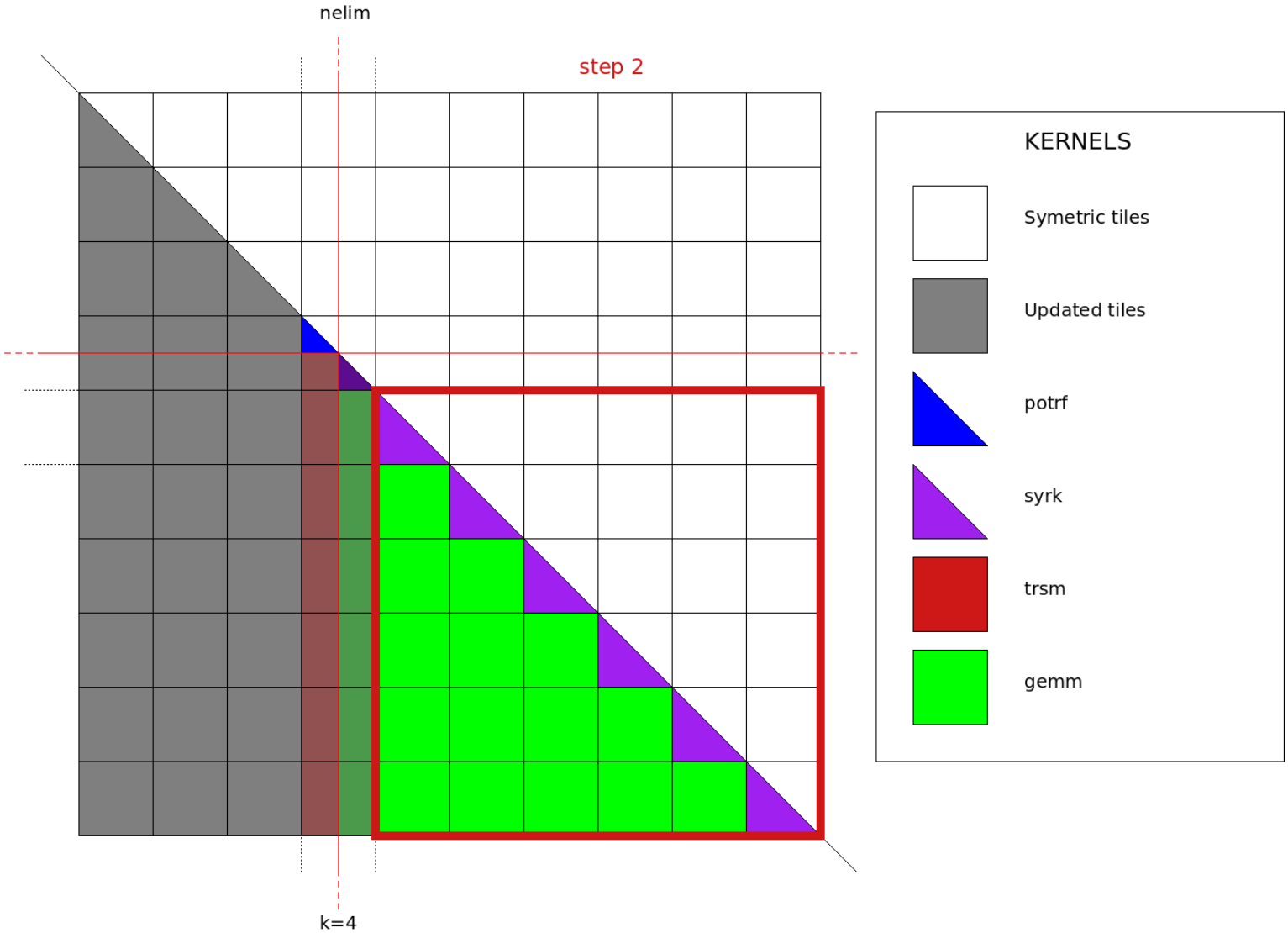
Validation (step 2)

- Finish factorization of "nelim" tile



Validation (step 2)

- Update the rest of the matrix



Validation (step 2)

- Factorize the rest of the matrix, as if NB | nelim