On the memory behavior of a multifrontal QR software

for multicore systems

Alfredo Buttari, CNRS-IRIT

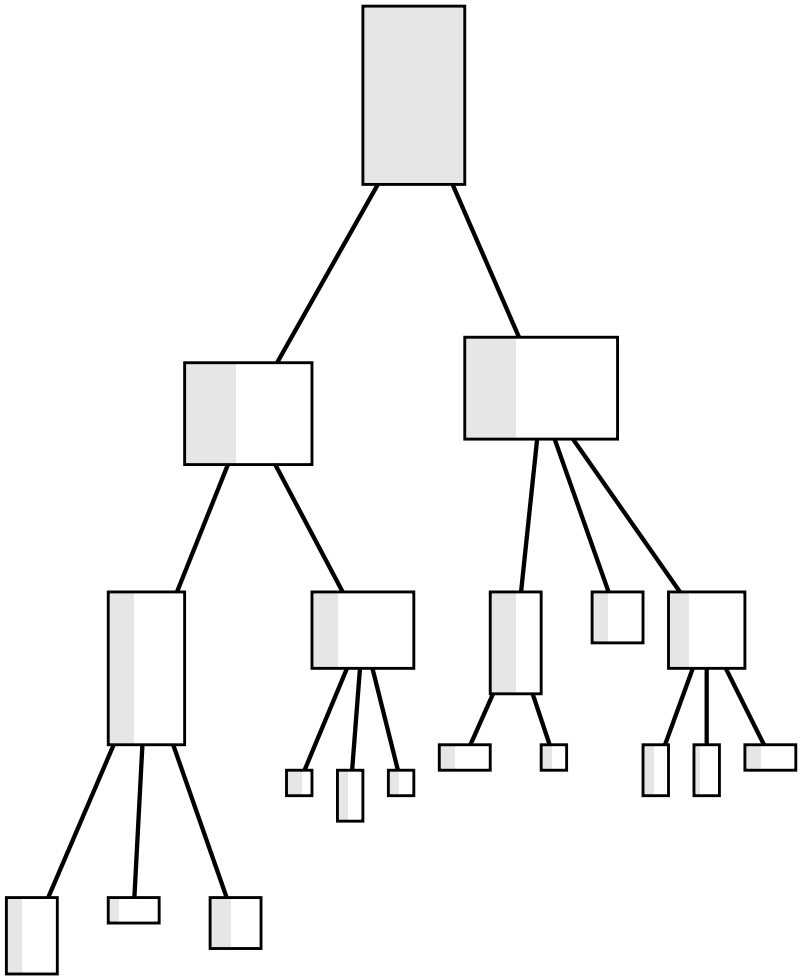
Toulouse, September 6-7, 2011

The multifrontal QR method

The Multifrontal QR for newbies

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

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



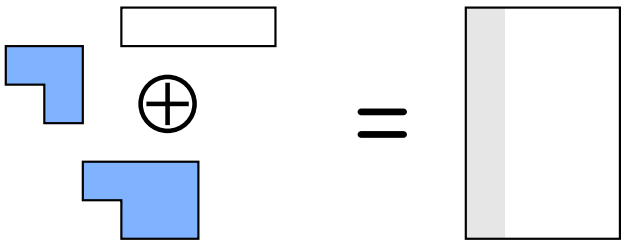
The Multifrontal QR for newbies

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

The Multifrontal QR for newbies

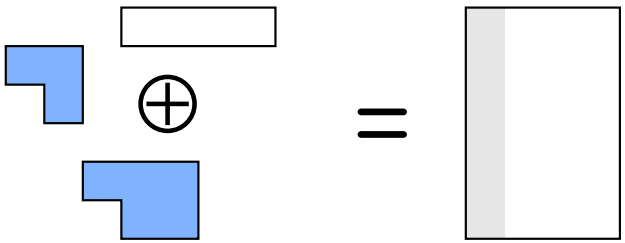
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

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

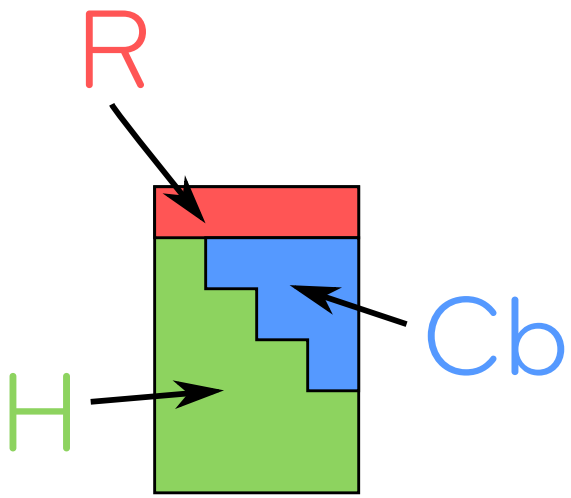


The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

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



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



Multifrontal QR, parallelism

The Multifrontal QR: parallelism

Two sources of parallelism are available in any multifrontal method:

Tree parallelism

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

Front parallelism

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

Parallelism: classical approach

The classical approach (Puglisi, Matstom, Davis)

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

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

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

Parallelism: a new approach

fine-grained, data-flow parallel approach

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

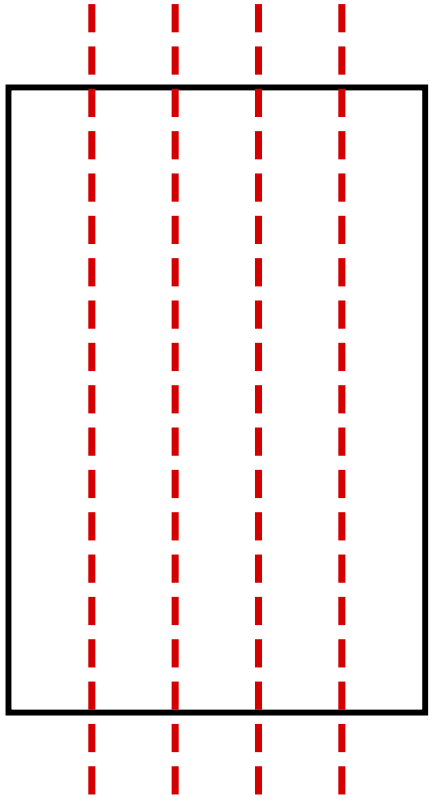
Both node and tree parallelism are handled the same way at any level of the tree.

Fine grained, asynchronous, parallel QR

Parallelism: a new approach

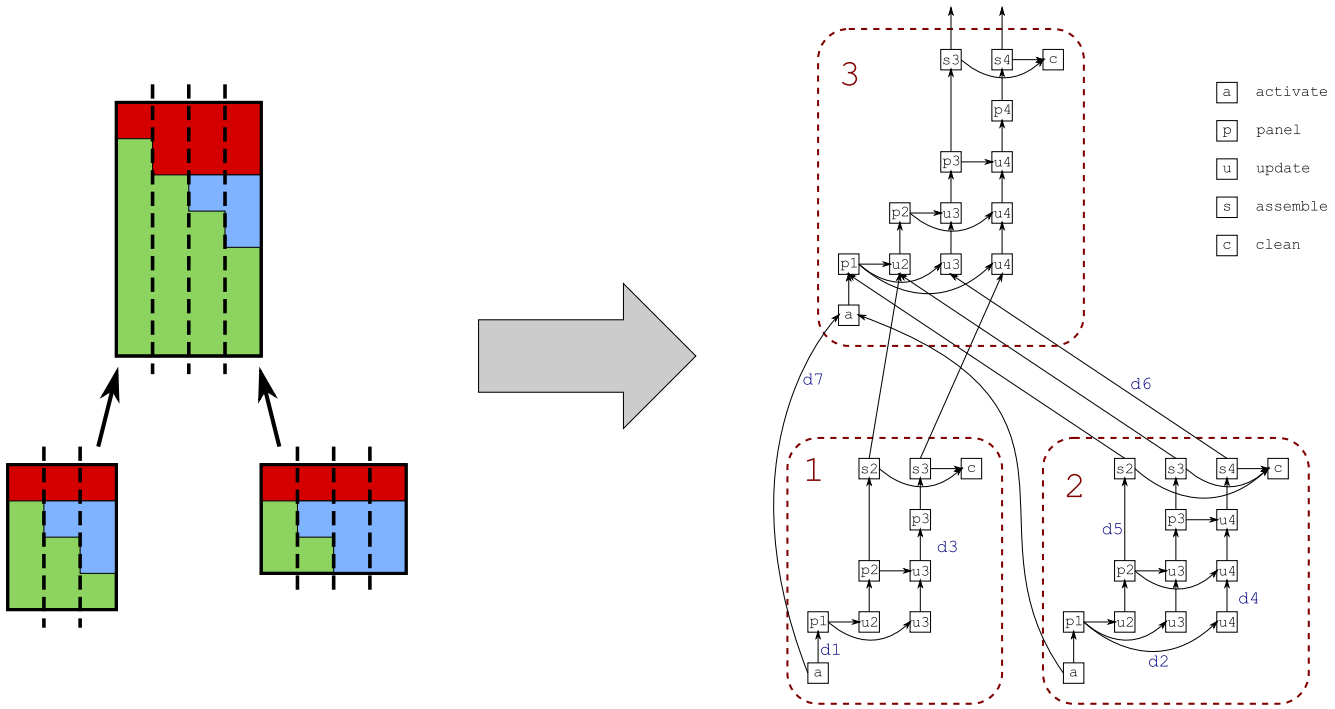
Fine-granularity is achieved through a 1-D block partitioning of fronts and the definition of five elementary operations:

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



Parallelism: a new approach

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



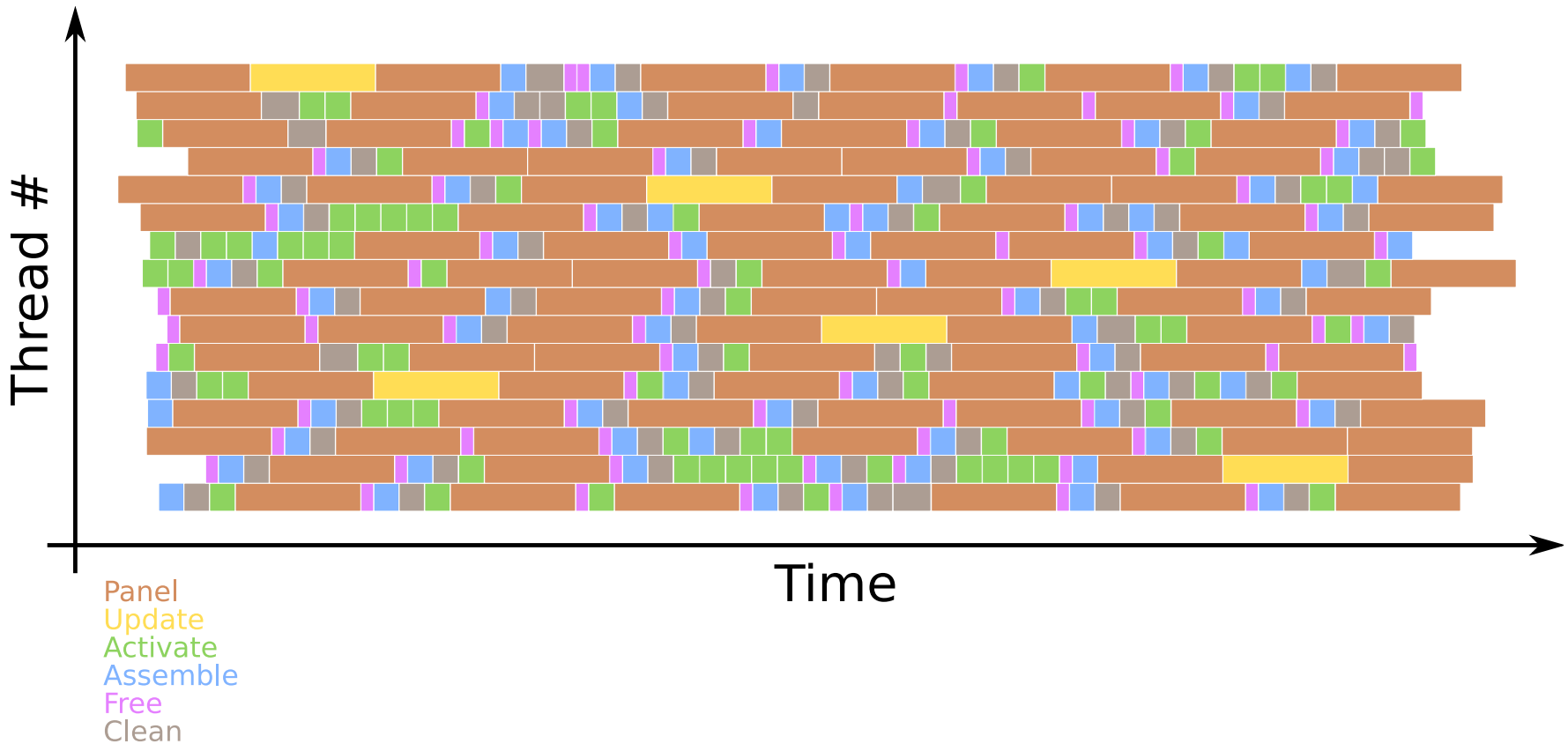
where the nodes represent tasks and edges the dependencies among them

Parallelism: a new approach

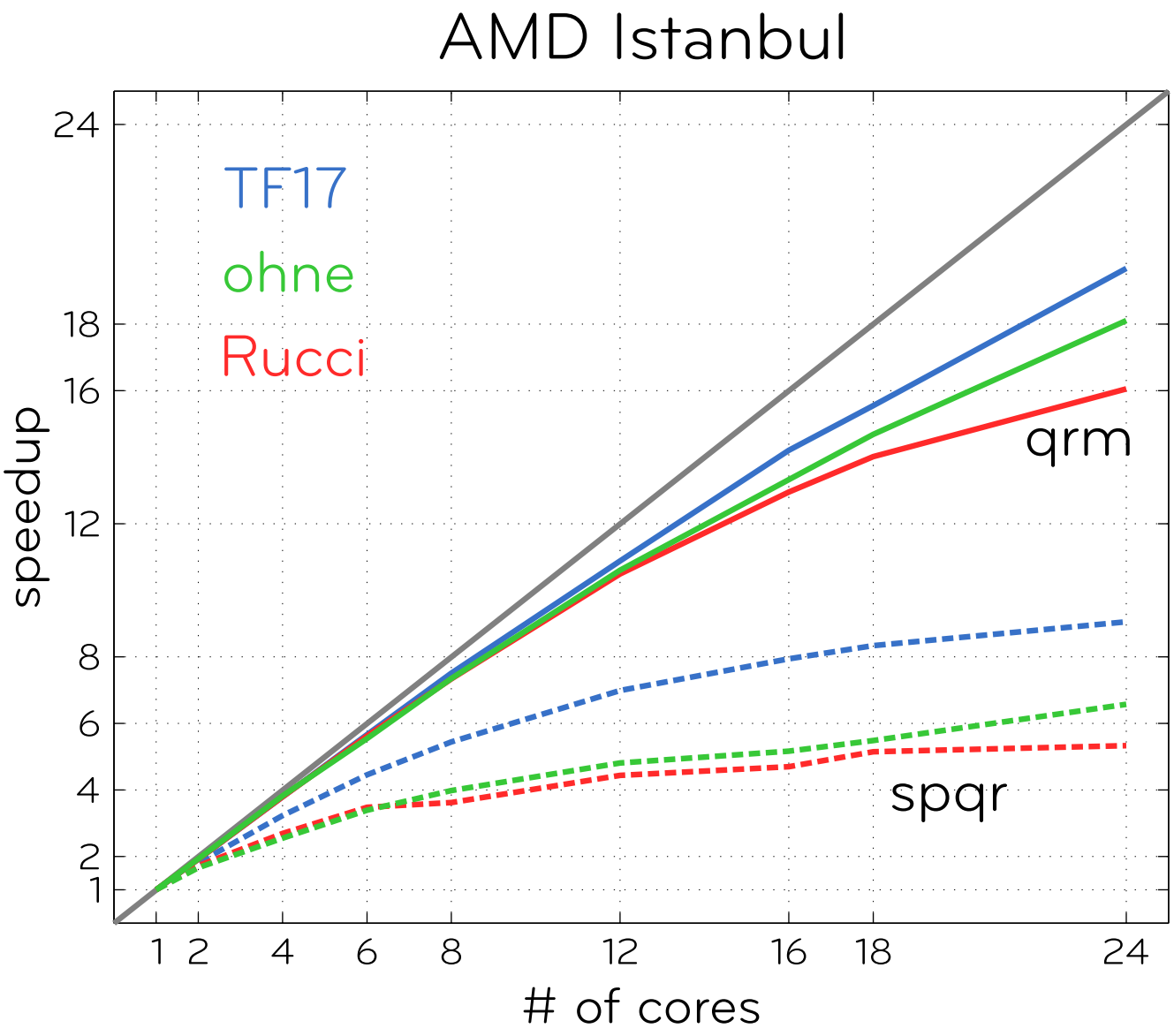
- d1: no other elementary operation can be executed on a front or on one of its block-columns until the front is not activated;
- d2: a block column can be updated with respect to a panel only if the corresponding panel factorization is completed;
- d3: the **panel** operation can be executed on block-column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

Parallelism: a new approach

Tasks are scheduled dynamically and asynchronously



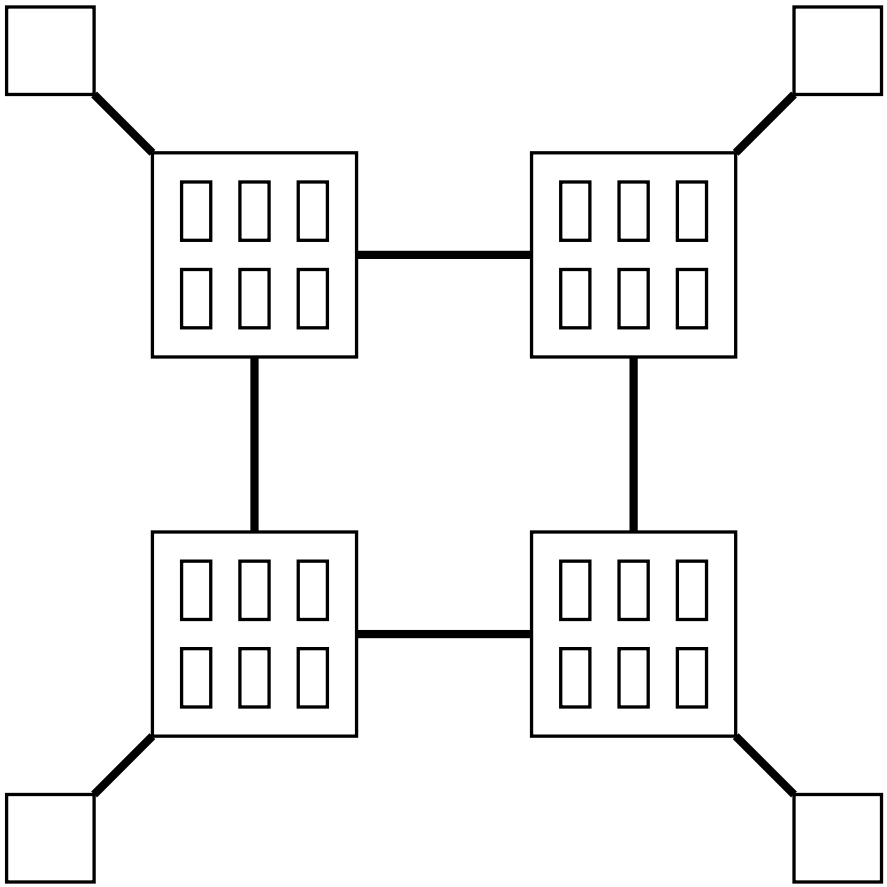
Results


AMD Istanbul
TF17
ohne
Rucci
qrm
spqr
speedup
# of cores
1
2
4
6
8
12
16
18
24

Tasks scheduling

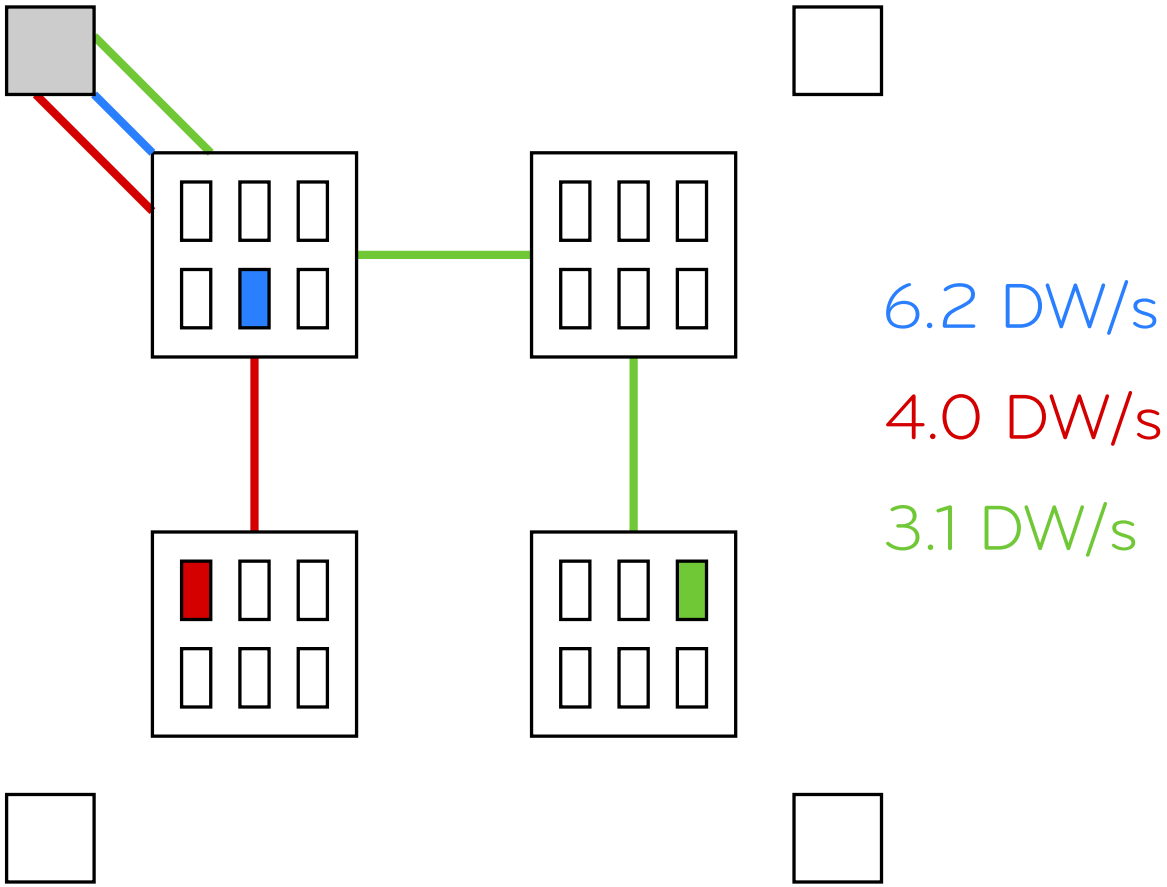
Target architecture

Our target architecture has four, hexa-core AMD Istanbul processors connected through HyperTransport links in a ring topology with a memory module attached to each of them:



Target architecture

Our target architecture has four, hexa-core AMD Istanbul processors connected through HyperTransport links in a ring topology with a memory module attached to each of them:



The bandwidth depends on the number of hops

Proximity scheduling

The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:

No front-to-core mapping is done (yet)!

Proximity scheduling

The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:

- At the moment when a thread activates a front it becomes its owner

No front-to-core mapping is done (yet)!

Proximity scheduling

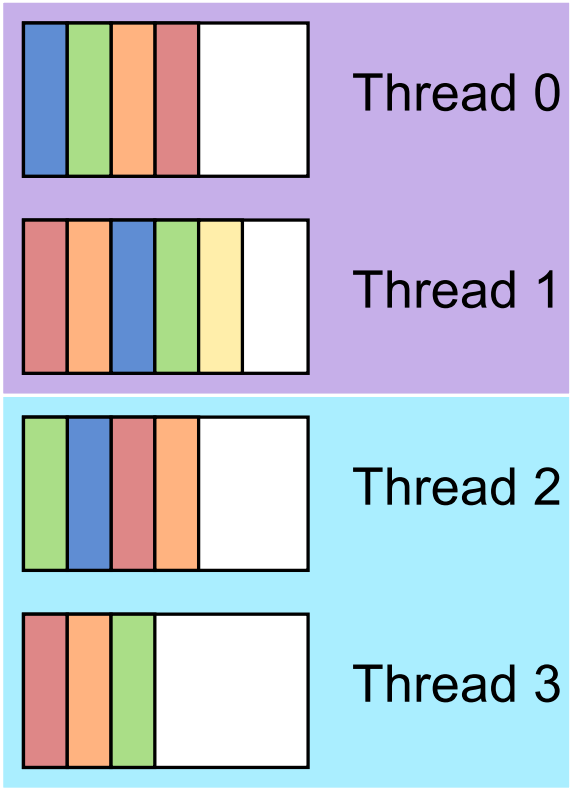
The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:

- At the moment when a thread activates a front it becomes its owner
- All the tasks related to a front will be pushed on the queue associated with its owner

No front-to-core mapping is done (yet)!

Proximity scheduling

The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:

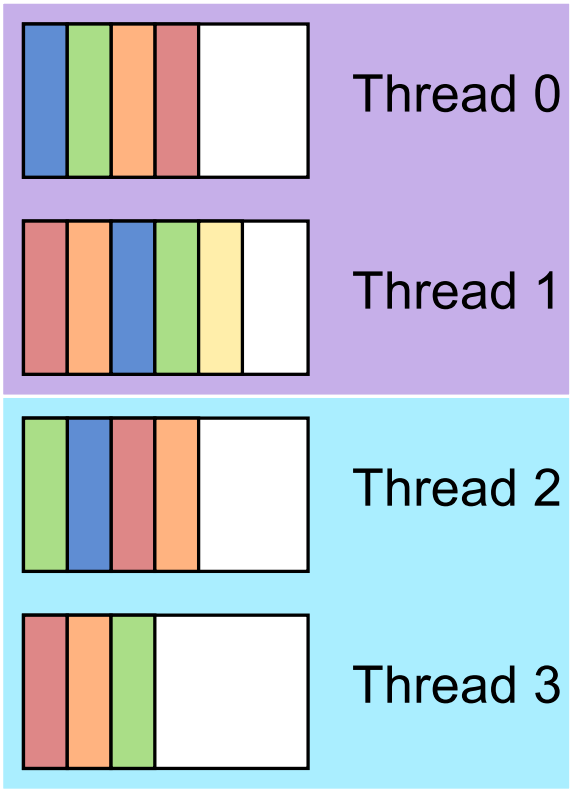


- At the moment when a thread activates a front it becomes its owner
- All the tasks related to a front will be pushed on the queue associated with its owner
- Work-stealing is used to feed threads that run out of tasks:

No front-to-core mapping is done (yet)!

Proximity scheduling

The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:

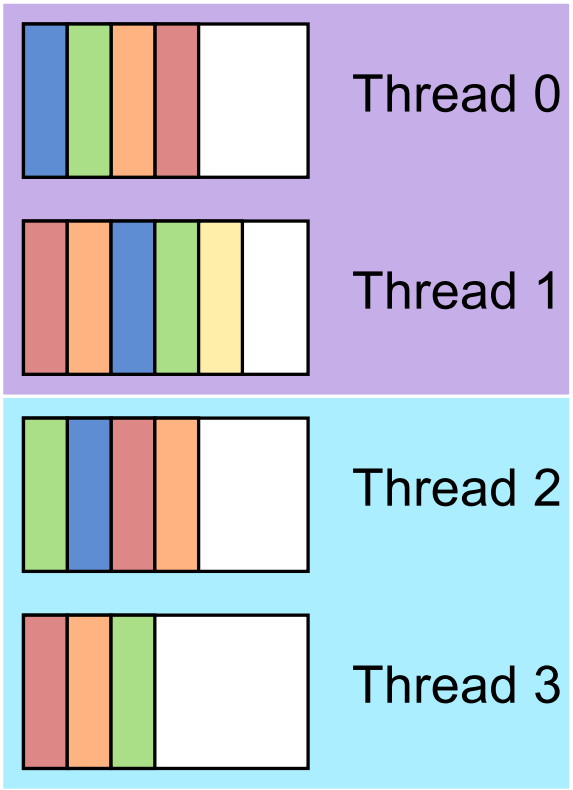


- At the moment when a thread activates a front it becomes its owner
- All the tasks related to a front will be pushed on the queue associated with its owner
- Work-stealing is used to feed threads that run out of tasks:
  - a thread will first try to steal tasks from neighbor queues...

No front-to-core mapping is done (yet)!

Proximity scheduling

The scheduling can be guided by a proximity criterion: a task should be executed by the core which is closest to the concerned data. This can be implemented through a system of task queues, one per thread/core:



- At the moment when a thread activates a front it becomes its owner
- All the tasks related to a front will be pushed on the queue associated with its owner
- Work-stealing is used to feed threads that run out of tasks:
  - a thread will first try to steal tasks from neighbor queues...
  - ...and then from any other queue

No front-to-core mapping is done (yet)!

Proximity scheduling

Implementing all this requires the ability to:

Proximity scheduling

Implementing all this requires the ability to:

- control the placement of threads: we have to bind each thread to a single core and prevent threads migrations. This can be done in a number of ways, e.g. by means of tools such as `hwloc` which allows thread pinning

Proximity scheduling

Implementing all this requires the ability to:

- control the placement of threads: we have to bind each thread to a single core and prevent threads migrations. This can be done in a number of ways, e.g. by means of tools such as `hwloc` which allows thread pinning
- control the placement of data: we have to make sure that one front physically resides on a specific NUMA module. This can be done with:
  - the first touch rule: the data is allocated close to the core that makes the first reference
  - `hwloc` or `numalib` which provide NUMA-aware allocators

Proximity scheduling

Implementing all this requires the ability to:

- control the placement of threads: we have to bind each thread to a single core and prevent threads migrations. This can be done in a number of ways, e.g. by means of tools such as `hwloc` which allows thread pinning
- control the placement of data: we have to make sure that one front physically resides on a specific NUMA module. This can be done with:
  - the first touch rule: the data is allocated close to the core that makes the first reference
  - `hwloc` or `numalib` which provide NUMA-aware allocators
- detect the architecture we have to figure out the memory/cores layout in order to guide the work stealing. This can be done with `hwloc`

Proximity scheduling: experiments

Experimental results show that proximity scheduling is good:

Matrix	Strat.	Time (s)
Rucci1	no loc.	155
	loc.	144
ohne2	no loc.	43
	loc.	39
lp_nug20	no loc.	89
	loc.	84

Why?

Proximity scheduling: experiments

Experimental results show that proximity scheduling is good:

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)
Rucci1	no loc.	155	2000
	loc.	144	1634
ohne2	no loc.	43	504
	loc.	39	375
lp_nug20	no loc.	89	879
	loc.	84	778

Why? Fewer Dwords transferred through the HyperTransport links, as shown by the number of occurrences of the PAPI `HYPERTRANSPORT_LINKx:DATA_DWORD_SENT` event.

Proximity scheduling: experiments

Experimental results show that proximity scheduling is good:

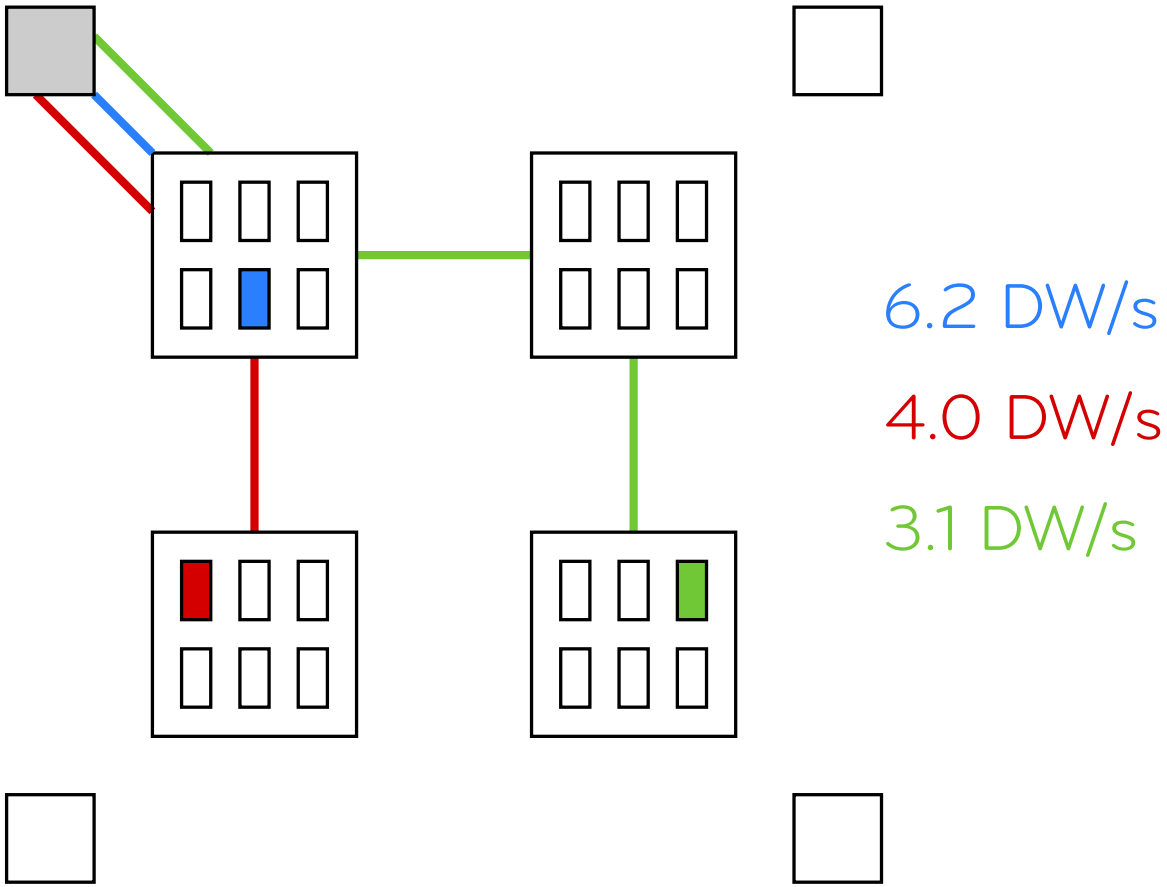
Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)
Rucci1	no loc.	155	2000
	loc.	144	1634
ohne2	no loc.	43	504
	loc.	39	375
lp_nug20	no loc.	89	879
	loc.	84	778

Why? Fewer Dwords transferred through the HyperTransport links, as shown by the number of occurrences of the PAPI `HYPERTRANSPORT_LINKx:DATA_DWORD_SENT` event.

But this not the whole story!

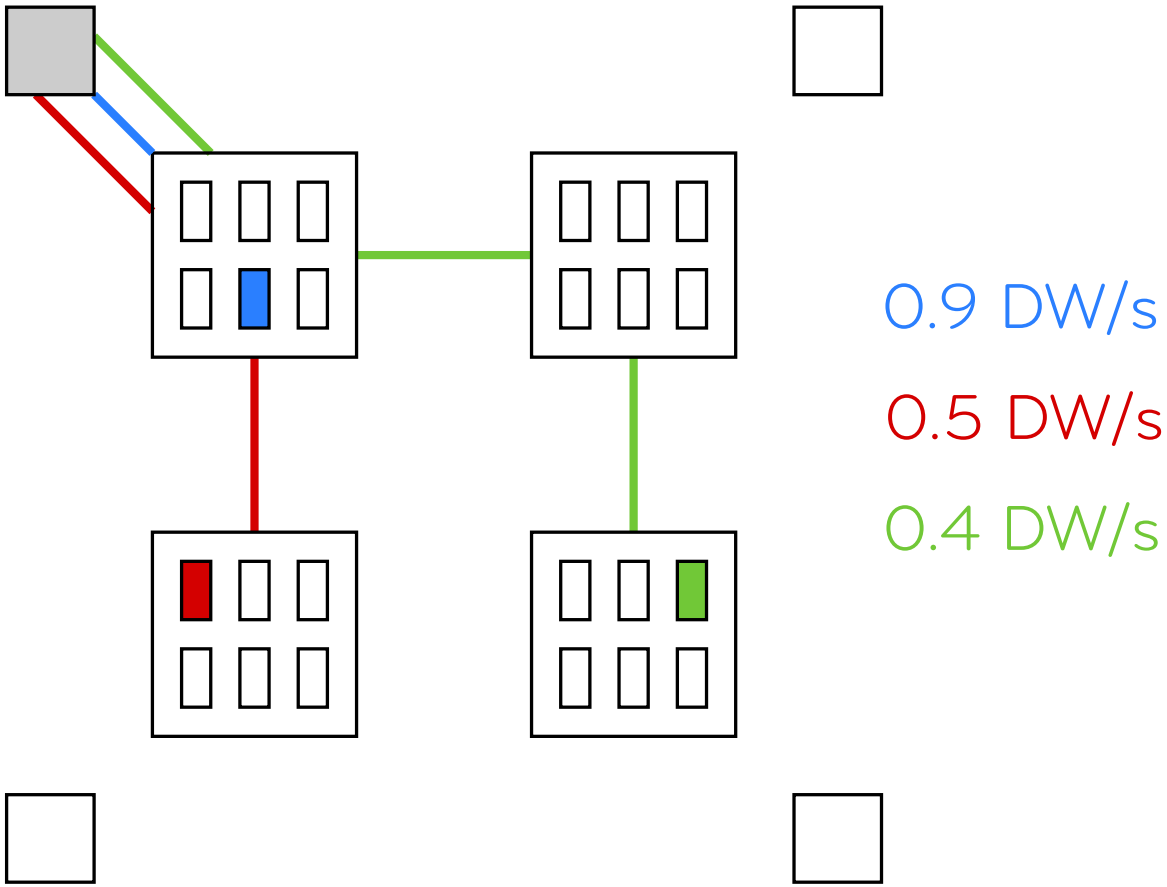
Target architecture

What happens when more threads are active at the same time on the system?



Target architecture

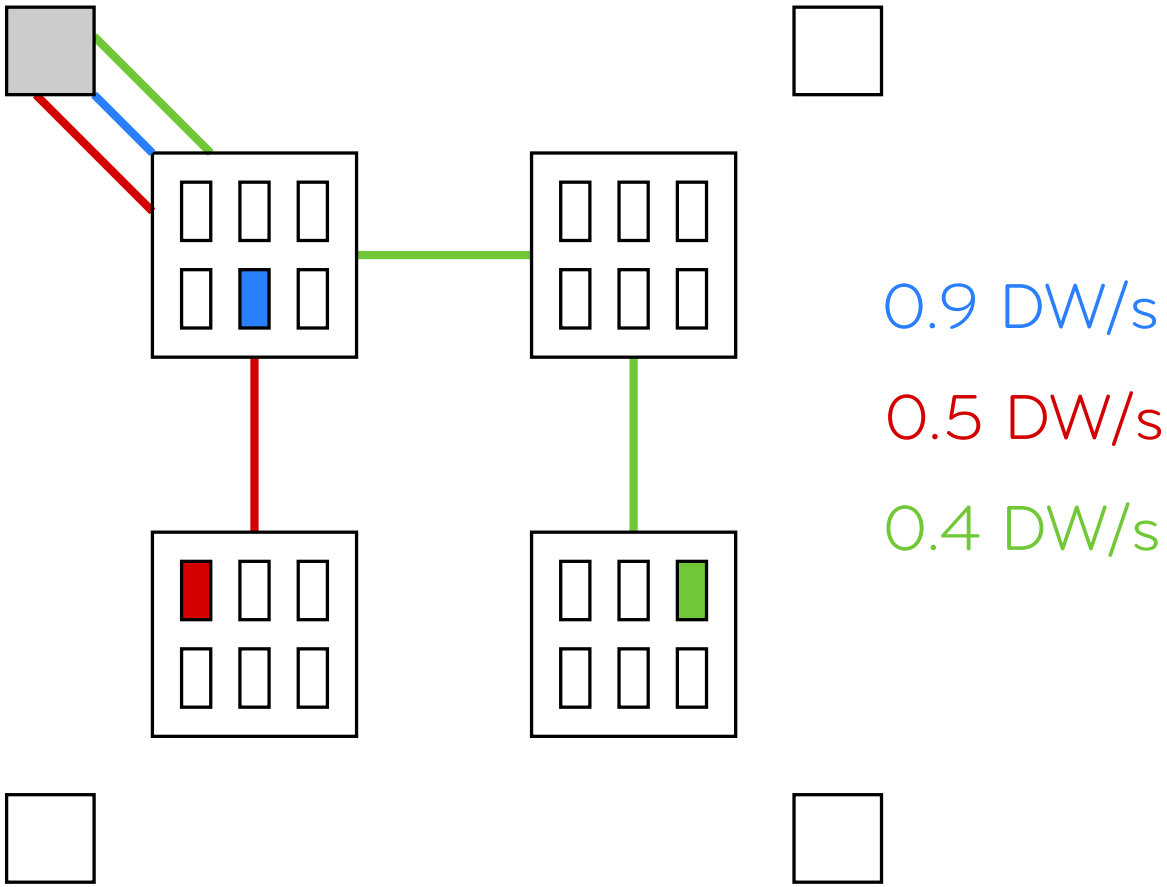
What happens when more threads are active at the same time on the system?



BW drops; why?

Target architecture

What happens when more threads are active at the same time on the system?

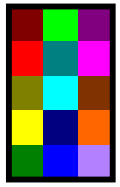


BW drops; why? memory conflicts!

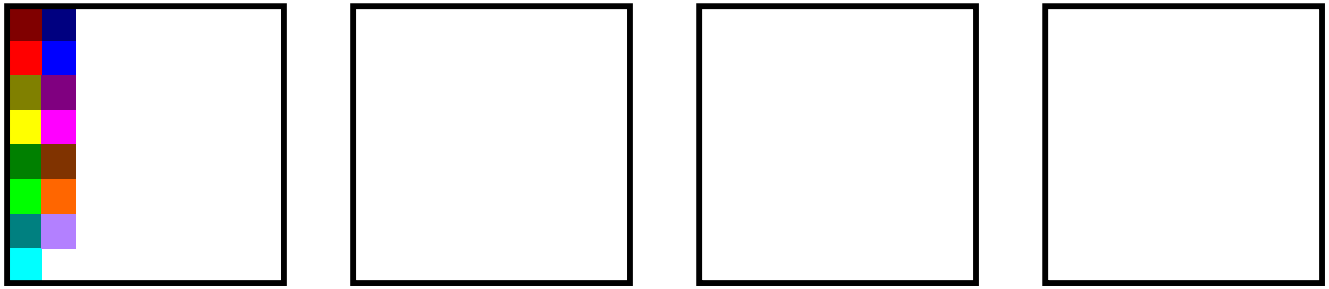
Experiments on conflicts

Should we do something to reduce conflicts?

Front



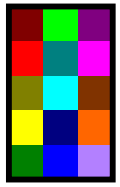
Normal allocation



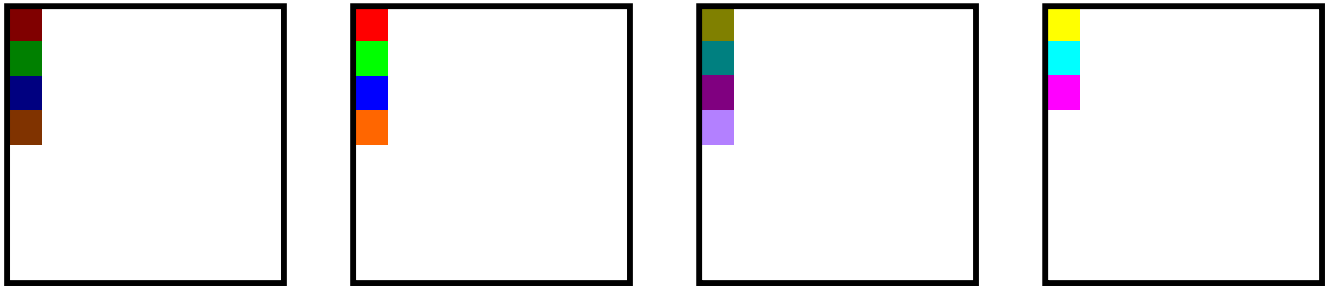
Experiments on conflicts

Should we do something to reduce conflicts?

Front



Interleaved allocation
(`numactl -i all`)



Memory interleaving should provide a more uniform distribution of data that (supposedly) reduces conflicts and increases the memory bandwidth

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)
Rucci1	no loc.	155	2000
	loc.	144	1634
ohne2	no loc.	43	504
	loc.	39	375
lp_nug20	no loc.	89	879
	loc.	84	778

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)
Rucci1	no loc.	155	2000
	loc.	144	1634
	r. r.	117	
ohne2	no loc.	43	504
	loc.	39	375
	r. r.	38	
lp_nug20	no loc.	89	879
	loc.	84	778
	r. r.	66	

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)
Rucci1	no loc.	155	2000
	loc.	144	1634
	r. r.	117	2306
ohne2	no loc.	43	504
	loc.	39	375
	r. r.	38	665
lp_nug20	no loc.	89	879
	loc.	84	778
	r. r.	66	985

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)	Conflicts on DCT ($\times 10^9$)
Rucci1	no loc.	155	2000	23.8
	loc.	144	1634	25.4
	r. r.	117	2306	19.7
ohne2	no loc.	43	504	6.63
	loc.	39	375	6.71
	r. r.	38	665	4.54
lp_nug20	no loc.	89	879	11.4
	loc.	84	778	11.9
	r. r.	66	985	6.81

Conflicts are `DRAM_ACCESSES_PAGE:DCTx_PAGE_CONFLICT` events measured by PAPI

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)	Conflicts on DCT ($\times 10^9$)
Rucci1	no loc.	155	2000	23.8
	loc.	144	1634	25.4
	r. r.	117	2306	19.7
ohne2	no loc.	43	504	6.63
	loc.	39	375	6.71
	r. r.	38	665	4.54
lp_nug20	no loc.	89	879	11.4
	loc.	84	778	11.9
	r. r.	66	985	6.81

Conflicts are `DRAM_ACCESSES_PAGE:DCTx_PAGE_CONFLICT` events measured by PAPI

Analogous behavior was found on the HSL `MA87` code.

Experiments on conflicts

Matrix	Strat.	Time (s)	Dwords on HT ($\times 10^9$)	Conflicts on DCT ($\times 10^9$)
Rucci1	no loc.	155	2000	23.8
	loc.	144	1634	25.4
	r. r.	117	2306	19.7
ohne2	no loc.	43	504	6.63
	loc.	39	375	6.71
	r. r.	38	665	4.54
lp_nug20	no loc.	89	879	11.4
	loc.	84	778	11.9
	r. r.	66	985	6.81

Conflicts are `DRAM_ACCESSES_PAGE:DCTx_PAGE_CONFLICT` events measured by PAPI

Analogous behavior was found on the HSL `MA87` code.

The answer is: yes, we should reduce conflicts (work in progress).

Conclusions

- fine-granularity, asynchronism and data-flow parallelism are tough (especially all together) but the effort is largely paid off
- a correct exploitation of the memory system is critical, especially in NUMA systems
- reducing memory transfers gives some performance improvement but...
- ...maybe it is not the most important thing: memory conflicts seem to be waaay more penalizing than data traffic. How do we solve this? Not clear for the moment. Maybe a careful front-to-memory mapping? Ideas and suggestions are very welcome

Thank you all!

Questions?