Sparse direct solvers on top of runtime systems

ANR SOLHAR

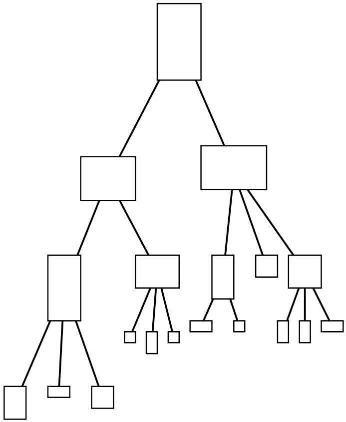
E. Agullo, G. Bosilca, A. Buttari, A. Guermouche and **F. Lopez**, Université de Toulouse-IRIT

Lunch Talk ICL 2014

The multifrontal QR method

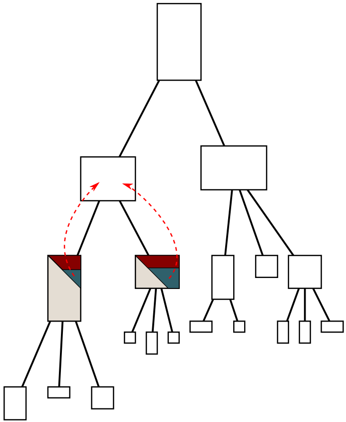
The multifrontal QR factorization is guided by a graph called *elimination tree*:

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

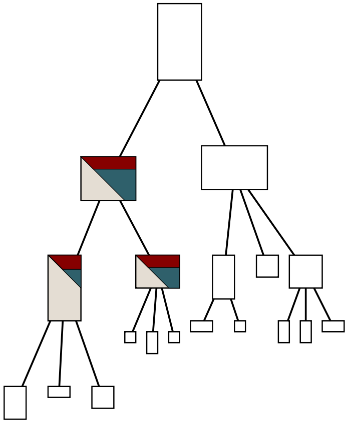
- assembly: coefficients from the original matrix associated with the pivots and *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix



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 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 father's front



The Multifrontal QR method

Notable differences with multifrontal LU:

- fronts are rectangular, either over or under-determined
- assembly operations are just copies (with lots of indirect addressing) and not sums. They can thus be done in any order (like in LU) but also in parallel (most likely not efficient because of false sharing issues)
- fronts are not full: they have a staircase structure. The zeroes in the lower-leftmost part can be ignored. This irregular structure makes the modeling of performance rather difficult
- fronts are completely factorized and not just partially. This makes the overall size of factors bigger and thus the active memory consumption less sensitive to the tree traversal
- contribution blocks are trapezoidal and note square

The Multifrontal QR method: parallelism

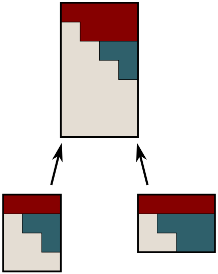
In the multifrontal methods we can distinguish two sources of parallelism:

Tree parallelism

Frontal matrices located in independent branches in the tree can be processed in parallel

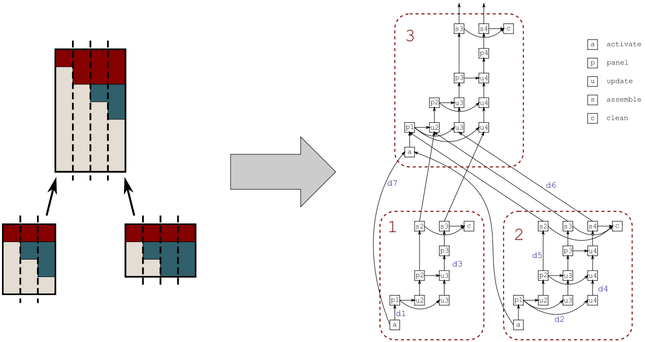
Node parallelism

Large frontal matrices factorization may be performed in parallel by multiple threads



The Multifrontal QR method in qr_mumps

Our baseline is the approach used in qr_mumps where the workload is expressed as a DAG of tasks defined through a 1D Block-column partitioning



In qr_mumps threading is implemented through OpenMP and scheduling of tasks is done “by hand”

Parallelism: a new approach

The scheduling is performed by a finely-tuned, hand-written code

- ▲ the fine-grained decomposition and the asynchronous/dynamic scheduling deliver high concurrency and much better performance compared to the classical approach (SPQR)
- ▼ the scheduler is not scalable (the search for ready tasks in the DAG is inefficient)...
- ▼ ... extremely difficult to maintain...
- ▼ ... and not really portable

Add new features in `qr_mumps`

We want to develop the following features in `qr_mumps`:

- 2D partitioning of frontal matrices (finer granularity allowing better parallelism) as 1D partitioning may not be adapted
  - most fronts are overdetermined
  - the problem is mitigated by concurrent front factorizations

- Exploit GPUs

- Memory-aware algorithms (perform factorization under a given memory constraint)
- Distributed memory architectures

Add new features in qr_mumps

We want to develop the following features in qr_mumps:

- 2D partitioning of frontal matrices (finer granularity allowing better parallelism) as 1D partitioning may not be adapted
  - most fronts are overdetermined
  - the problem is mitigated by concurrent front factorizations
  - ▲ more concurrency
  - ▼ more complex dependencies, more tasks
- Exploit GPUs
  - ▼ memory transfers, CUDA kernels management
- Memory-aware algorithms (perform factorization under a given memory constraint)
- Distributed memory architectures
  - ▼ MPI layer

All these problems may be overcome by using runtime system

STF vs PTG models

STF vs PTG models

The Sequential Task Flow (STF) model in StarPU:

- The parallel corresponds to the sequential one except that operations are not executed but submitted to the system in the form of tasks
- Depending on data access in tasks and the order of submission, the runtime infers dependencies among them and builds a DAG

Drawbacks of this model:

- The DAG is entirely unrolled in the runtime: limited scalability

STF vs PTG models

The Parametrized Task Graph (PTG) model in PaRSEC:

- The DAG is represented with a compact format where the different type of tasks are defined (domain of definition, CPU/GPU implementation) as well as their dependencies wrt other tasks (input/output data)
- On task completion, the DAG is partially unrolled following released data dependencies

Drawbacks of this model:

- programming model less intuitive than STF

STF vs PTG models

The Parametrized Task Graph (PTG) model in PaRSEC:

- The DAG is represented with a compact format where the different type of tasks are defined (domain of definition, CPU/GPU implementation) as well as their dependencies wrt other tasks (input/output data)
- On task completion, the DAG is partially unrolled following released data dependencies

Drawbacks of this model:

- programming model less intuitive than STF

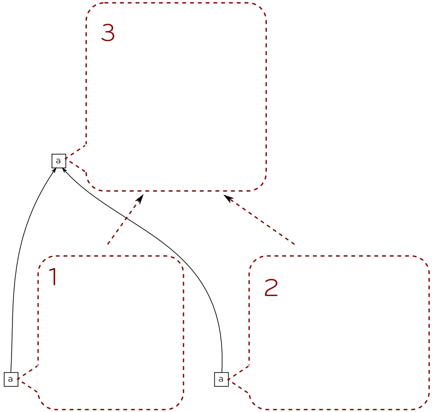
Objective

Develop a PaRSEC version of `qr_mumps` following the PTG model and evaluate its effectiveness on a single-node, multicore systems

PaRSEC multifrontal QR

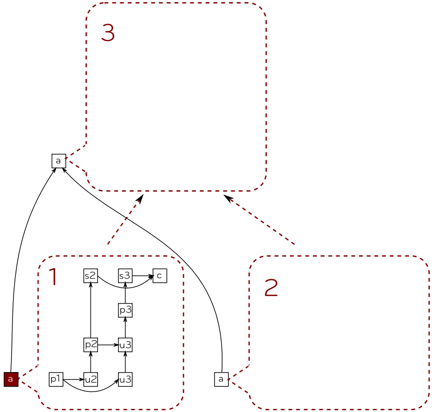
PaRSEC Multifrontal QR

- The elimination tree is represented in a main JDF
- The front factorization is represented in separate JDFs
  - 1D block partitioning
  - 2D block partitioning (not necessarily square) with flat, binary (communication avoiding) or hybrid panel reduction trees
- Upon activation (allocating memory and initializing structures), the DAG corresponding to the front factorization is spawned in PaRSEC



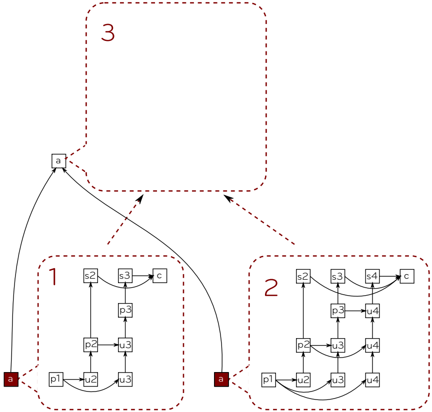
PaRSEC Multifrontal QR

- The elimination tree is represented in a main JDF
- The front factorization is represented in separate JDFs
  - 1D block partitioning
  - 2D block partitioning (not necessarily square) with flat, binary (communication avoiding) or hybrid panel reduction trees
- Upon activation (allocating memory and initializing structures), the DAG corresponding to the front factorization is spawned in PaRSEC



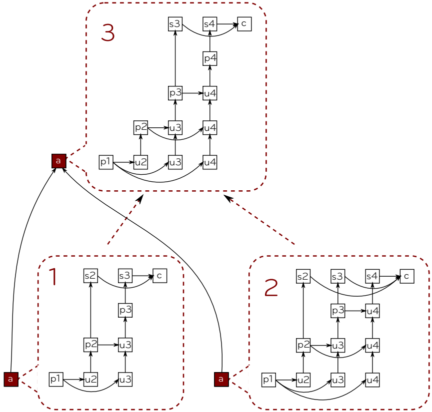
PaRSEC Multifrontal QR

- The elimination tree is represented in a main JDF
- The front factorization is represented in separate JDFs
  - 1D block partitioning
  - 2D block partitioning (not necessarily square) with flat, binary (communication avoiding) or hybrid panel reduction trees
- Upon activation (allocating memory and initializing structures), the DAG corresponding to the front factorization is spawned in PaRSEC



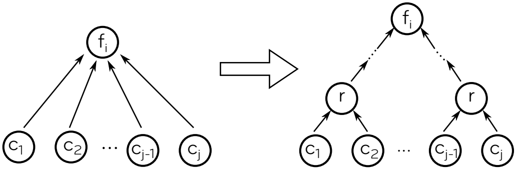
PaRSEC Multifrontal QR

- The elimination tree is represented in a main JDF
- The front factorization is represented in separate JDFs
  - 1D block partitioning
  - 2D block partitioning (not necessarily square) with flat, binary (communication avoiding) or hybrid panel reduction trees
- Upon activation (allocating memory and initializing structures), the DAG corresponding to the front factorization is spawned in PaRSEC

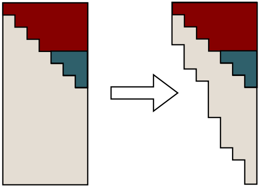


PaRSEC Multifrontal QR

- Elimination tree and assembly operations have an irregular input/output data-flow: tricky to express in the JDF format



- Fronts matrices have a sparse structure (staircase): the corresponding factorization DAG must be adapted from dense kernels

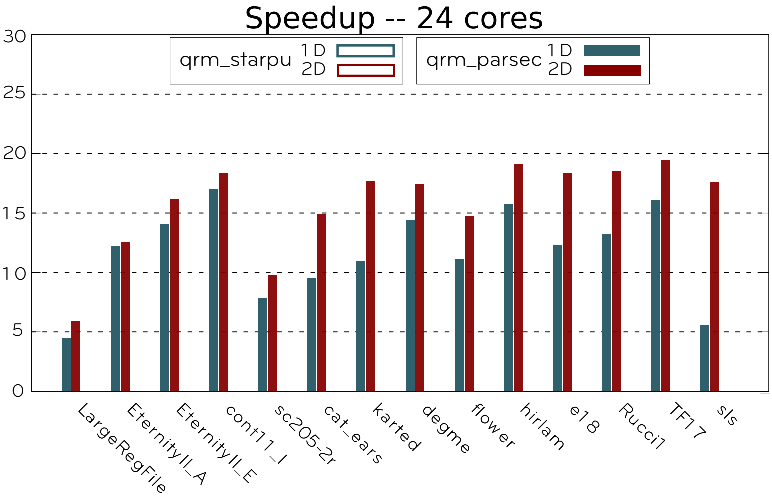


Experimental results

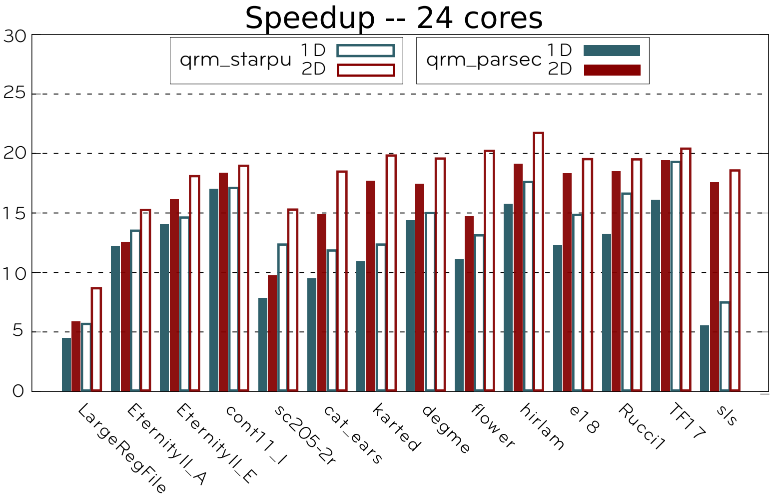
#	Matrix	Gflops	Ordering
1	LargeRegFile	19	Metis
2	EternityII_A	39	Metis
3	EternityII_E	107	Metis
4	cont11_l	112	Metis
5	sc205-2r	160	Metis
6	cat_ears_4_4	184	Metis
7	karted	335	Metis
8	degme	558	Metis
9	flower_7_4	724	Metis
10	hirlam	1112	Metis
11	e18	1286	Metis
12	Rucci1	5179	Metis
13	TF17	15663	Metis
14	sls	26363	Metis

- **System 1**:
  - IBM x3755
  - AMD Opteron Processor 8431 @ 2.4 GHz, 4×6 cores
  - 72 GB memory (NUMA)

PaRSEC Multifrontal QR: results

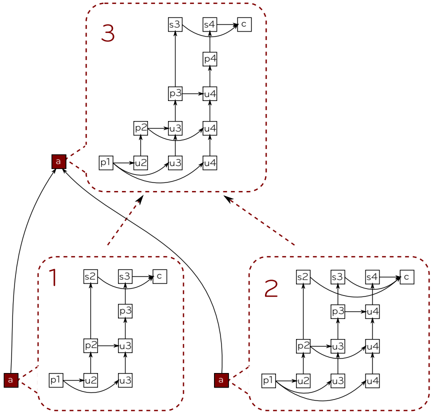


PaRSEC Multifrontal QR: results



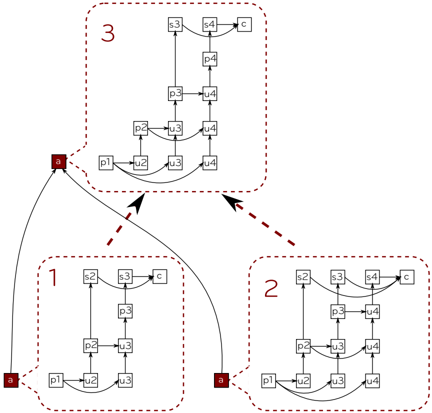
PaRSEC Multifrontal QR: results

- In the etree the parent-child dependencies are not finely managed resulting in poorer pipeline in the case of qrm_parsec



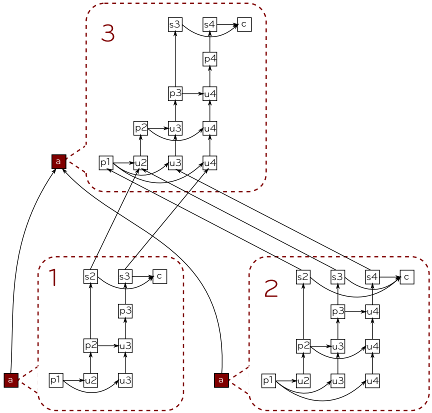
PaRSEC Multifrontal QR: results

- In the etree the parent-child dependencies are not finely managed resulting in poorer pipeline in the case of qrm_parsec



PaRSEC Multifrontal QR: results

- In the etree the parent-child dependencies are not finely managed resulting in poorer pipeline in the case of qrm_parsec
- Due to limitations in PaRSEC (not in the PTG model) it is not currently possible to achieve a pure data-flow parallelism



STF vs PTG: Front partitioning

How can we take advantage of the PTG model?

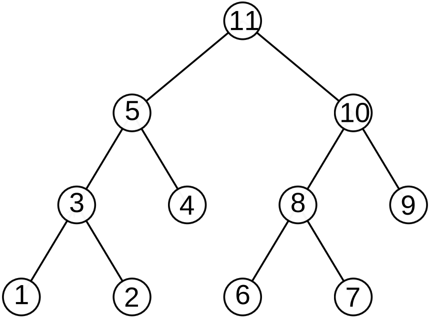
- The STF model allows a static approach: front partitioning occurs at the beginning of the factorization
- The PTG model allows a dynamic approach: front partitioning occurs upon front activation

In the PTG model the front partitioning may be decided depending on the context of execution

STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

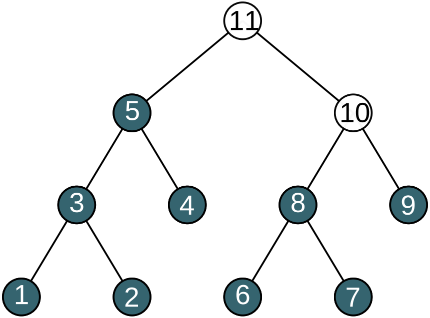
- Tree parallelism at the bottom of the tree: coarse grain partitioning (1D partitioning or rectangular tiles)
  - better kernel efficiency
  - less tasks: less scheduling overhead



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

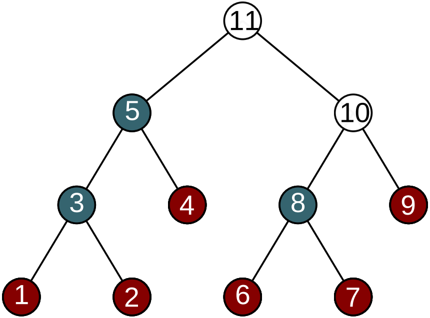
- Tree parallelism at the bottom of the tree: coarse grain partitioning (1D partitioning or rectangular tiles)
  - better kernel efficiency
  - less tasks: less scheduling overhead



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

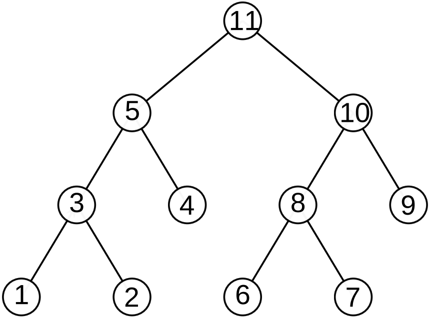
- Tree parallelism at the bottom of the tree: coarse grain partitioning (1D partitioning or rectangular tiles)
  - better kernel efficiency
  - less tasks: less scheduling overhead



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

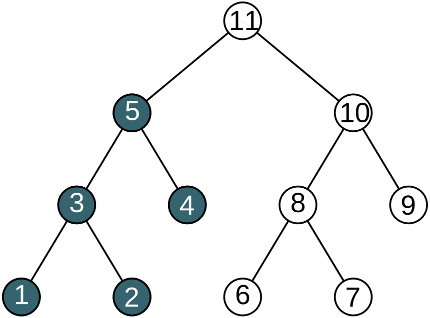
- Node parallelism at top of the tree or when reaching a memory constraint: fine grain partitioning
  - more parallelism
  - better pipeline



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

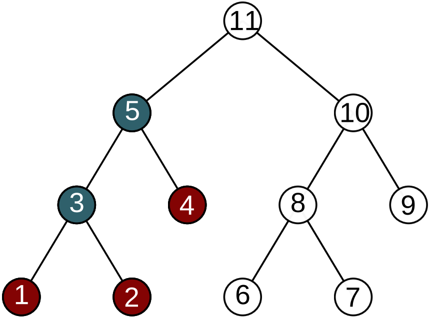
- Node parallelism at top of the tree or when reaching a memory constraint: fine grain partitioning
  - more parallelism
  - better pipeline



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

- Node parallelism at top of the tree or when reaching a memory constraint: fine grain partitioning
  - more parallelism
  - better pipeline



STF vs PTG: Front partitioning

How can we adapt the front partitioning depending on the context of execution?

- Extremely challenging to apply these rules in practice:
  - Huge search space for parameters
    - Tile dimensions
    - Inner blocking sizes
    - Panel reduction trees
  - Take into account the sparse structure of frontal matrices (staircase structure)

Conclusions and future work

Conclusions on PaRSEC

- More challenging to use than other runtimes systems
- Potentially more scalable
- Some features should be added PaRSEC to enhance the current version of qr_parsec

Ongoing and Future work

- Use GPUs with PaRSEC
- Distributed-memory architecture

?

Thanks for the welcome at ICL!
Questions?