QR_MUMPS: A RUNTIME-BASED SEQUENTIAL TASK FLOW PARALLEL SOLVER

E. Agullo, A. Buttari, A. Guermouche, F. Lopez and I. Masliah

Journée Runtime, 20-01-2017 , Bordeaux

THE MULTIFRONTAL QR FACTORIZATION

Sparse linear systems

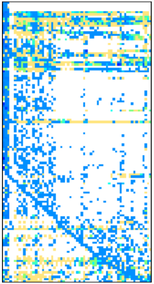
Many applications from physics, engineering, chemistry, geodesy, etc, require the solution of a linear system like

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

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

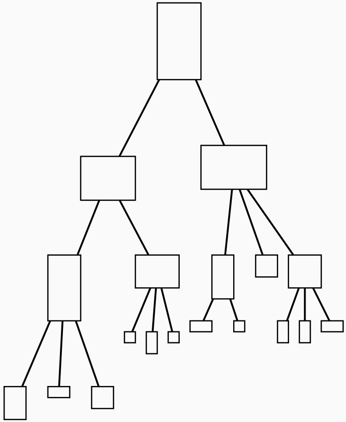
A sparse matrix is mostly filled with zeros:

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



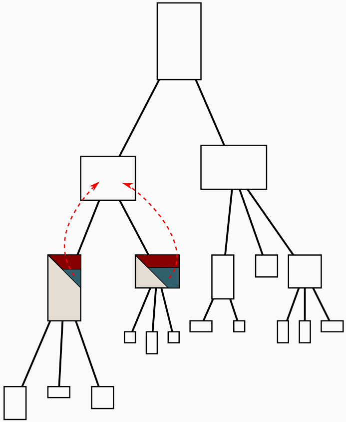
The original multifrontal method by Duff & Reid '83 can be extended to QR factorization of sparse matrices.
This method is guided by a graph called *elimination tree*:

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



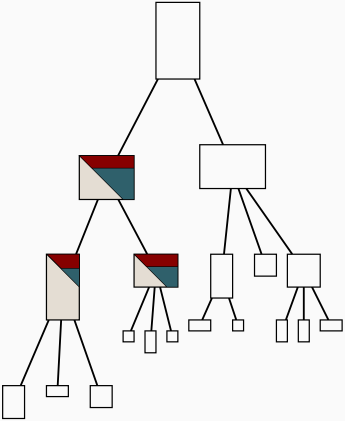
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- 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.

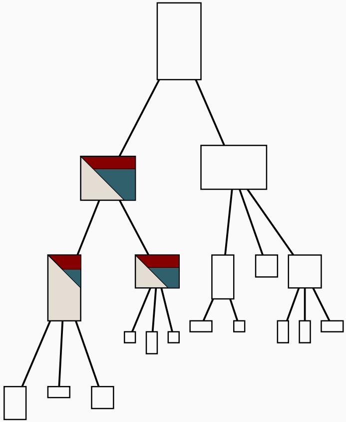


The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- assembly: coefficients from the original matrix associated with the pivots and *contribution blocks* produced by the treatment of the child nodes are stacked to form the frontal matrix.
- factorization: the k pivots are eliminated through a complete dense QR factorization of the frontal matrix. As a result we get:
  - part of the global R and Q factors.
  - a triangular *contribution block* that will be assembled into the father's front.

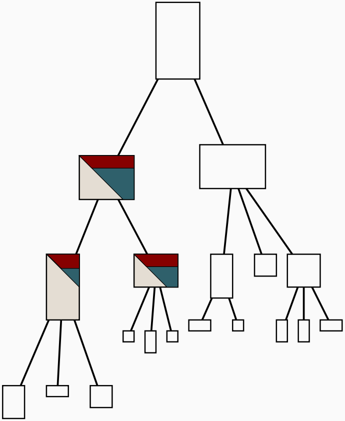


Typically two sources of parallelism are exploited in the multifrontal method



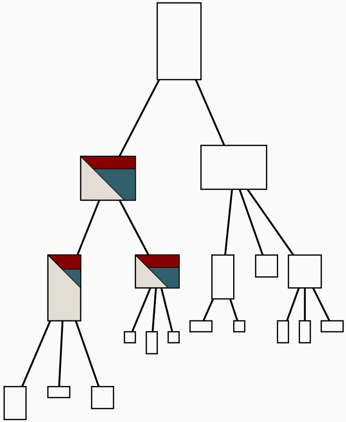
Typically two sources of parallelism are exploited in the multifrontal method

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

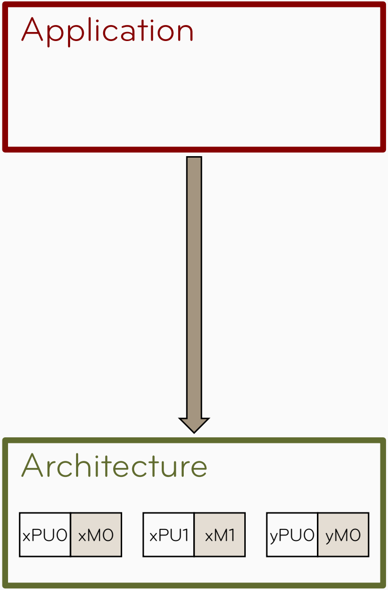


Typically two sources of parallelism are exploited in the multifrontal method

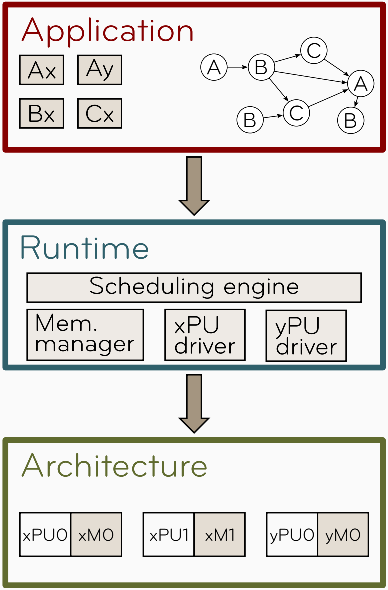
- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.
- node-level parallelism: large frontal matrices factorization may be performed in parallel by multiple threads.



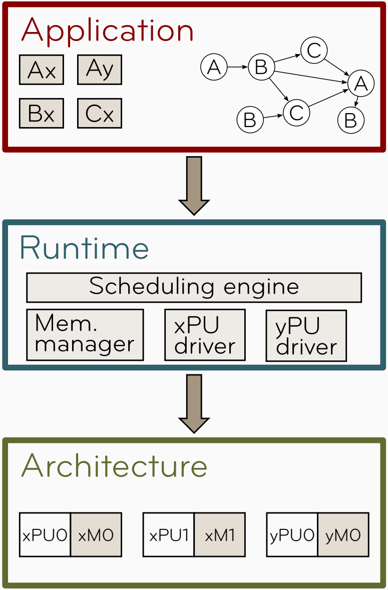
RUNTIME SYSTEMS



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



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



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

The Sequential Task Flow model: a simple example

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

The Sequential Task Flow model: a simple example

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
```

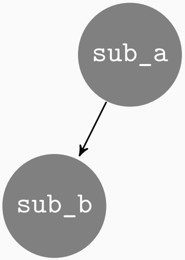
The Sequential Task Flow model: a simple example

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
submit(sub_b,x:R);
```

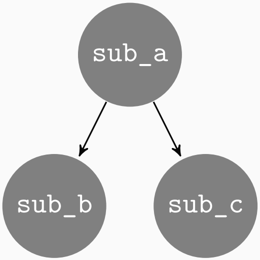


Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
submit(sub_b,x:R);
submit(sub_c,y:R);
```



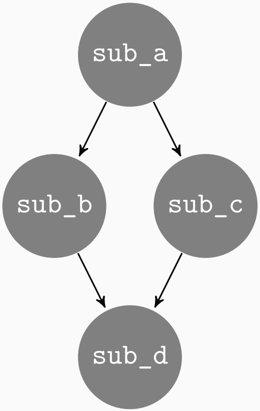
The Sequential Task Flow model: a simple example

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
submit(sub_b,x:R);
submit(sub_c,y:R);
submit(sub_d,x:RW,y:RW);
```



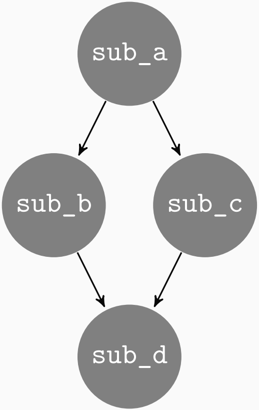
The Sequential Task Flow model: a simple example

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
submit(sub_b,x:R);
submit(sub_c,y:R);
submit(sub_d,x:RW,y:RW);
wait_tasks_completion( );
```



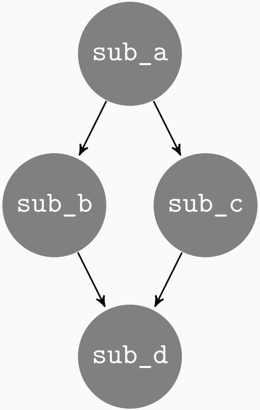
`sub_b` and `sub_c` can be executed in parallel.

Sequential code

```
sub_a(x,y); // R and W x and y
sub_b(x);   // R x
sub_c(y);   // R y
sub_d(x,y); // R and W x and y
```

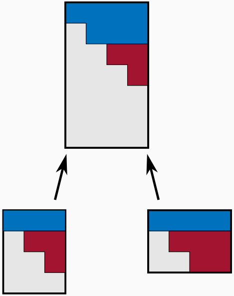
Equivalent STF code

```
submit(sub_a,x:RW,y:RW);
submit(sub_b,x:R);
submit(sub_c,y:R);
submit(sub_d,x:RW,y:RW);
wait_tasks_completion( );
```



`sub_b` and `sub_c` can be executed in parallel. If `sub_a` is executed on CPU and `sub_b` on GPU, `x` will be automatically transferred.

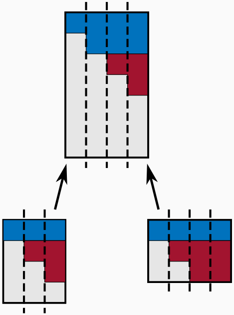
STF MULTIFRONTAL QR



```
forall fronts f in topological order

   ! compute front structure
   call activate(f)
   ! allocate and initialize front
   call init(f)

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

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

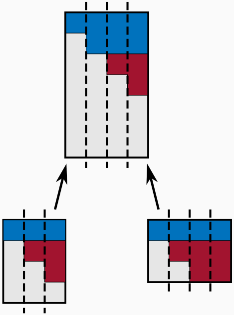
Sequential multifrontal *QR* code

The task-based multifrontal QR factorization



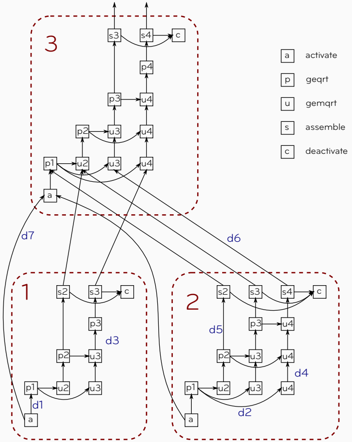
```
do f=1, nfronts ! in postorder
   ! compute front structure
   call activate(f)
   ! allocate and initialize front
   call init(f)

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

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

Sequential multifrontal *QR* code with 1D block partitioning



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

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

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

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



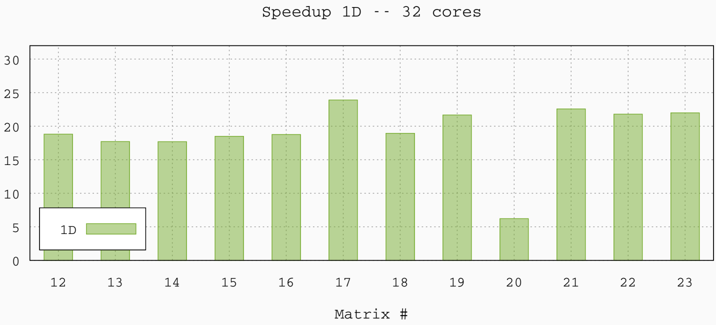

```
do f=1, nfronts ! in postorder
   ! compute structure and register handles
   call activate(f)
   ! allocate and initialize front
   call submit(init, f:RW)

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

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

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

Matrices from the UF SParse Matrix Collection:

#	Matrix	Mflops	Ordering
12	hirlam	1384160	SCOTCH
13	flower_8_4	2851508	SCOTCH
14	Rucci1	5671282	SCOTCH
15	ch8-8-b3	10709211	SCOTCH
16	GL7d24	16467844	SCOTCH
17	neos2	20170318	SCOTCH
18	spal_004	30335566	SCOTCH
19	n4c6-b6	62245957	SCOTCH
20	sls	65607341	SCOTCH
21	TF18	194472820	SCOTCH
22	lp_nug30	221644546	SCOTCH
23	mk13-b5	259751609	SCOTCH

ADA supercomputer at IDRIS: Intel Sandy Bridge E5-4650 @ 2.7 GHz, 4×8 cores

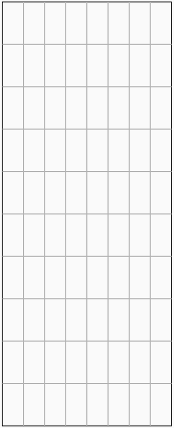


The task-based multifrontal method, implemented with a STF parallel model on top of StarPU offers good speedups on 32 cores:

- speedup increases with problem size with very low speedup for some problem such as matrix # 20
- we use a detailed performance analysis to determine the limiting factors of the STF 1D approach

1D partitioning is not good for (strongly) overdetermined matrices:

- ▼ Most fronts are overdetermined
- ▲ The problem is mitigated by concurrent front factorizations



- 2D block partitioning (not necessarily square)
- Communication avoiding algorithms
- ▲ More concurrency
- ▼ More complex dependencies
- ▼ Many more tasks (higher runtime overhead)
- ▼ Finer task granularity (less kernel efficiency)

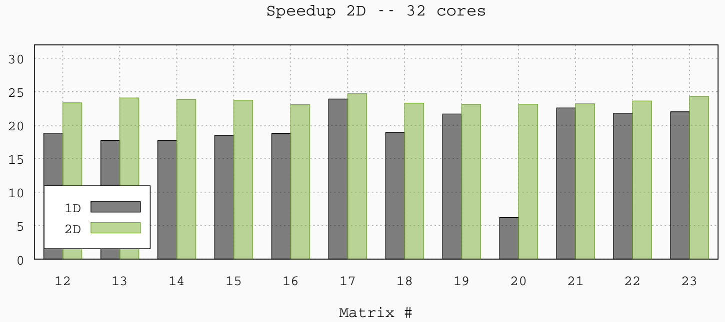
Thanks to the simplicity of the STF programming model it is possible to plug in 2D methods for factorizing the frontal matrices with a relatively moderate effort

1D PARTITIONING FRONT FACTORIZATION

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

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

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

2D partitioning + CA front factorization

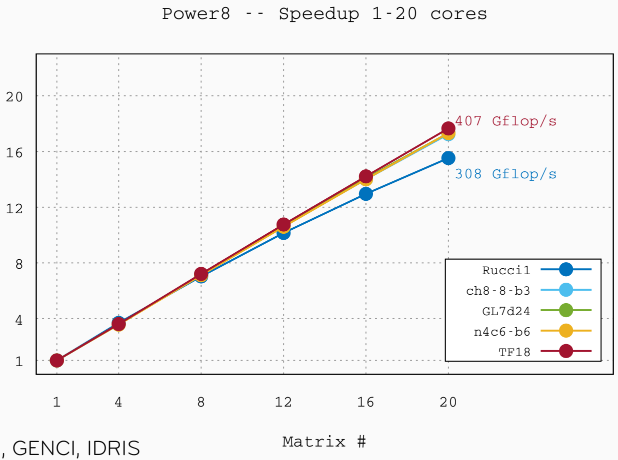
```
do f=1, nfronts                                     ! in postorder
  call activate(f)                                  ! activate front
  call submit(init, f:RW)                           ! init front

  do c=1, f%nchildren                               ! for all the children of f
    do i=1,c%m
      do j=1,c%n
        call submit(assemble, c(i,j):R, f:RW) ! assemble block(i,j) of c
      end do
    end do
    call submit(deactivate, c:RW)                   ! Deactivate child
  end do

  ca_facto: do k=1, min(f%m,f%n)
    do s=0, log2(f%m-k+1)
      do i = k, f%n, 2**s
        if(s.eq.0) then
          call submit(_geqrt, f(i,k):RW)
          do j=k+1, f%n
            call submit(_gemqrt, f(i,k):R, f(i,j):RW)
          end do
        else
          l = i+2**(s-1)
          call submit(_tpqrt, f(i,k):RW, f(l,k):RW)
          do j=k+1, front%n
            call submit(_tpmqrt, f(l,k):R, f(i,j):RW, f(l,j):RW)
          end do
        end if
      end do
    end do
  end do ca_facto
end do
call wait_tasks_completion()                        ! wait for the tasks to be executed
```



The scalability of the task-based multifrontal method is enhanced by the the introduction of 2D CA algorithms:

- Speedups are uniform for all tested matrices.
- We perform a comparative performance analysis wrt to the 1D case to show the benefits of the 2D scheme.

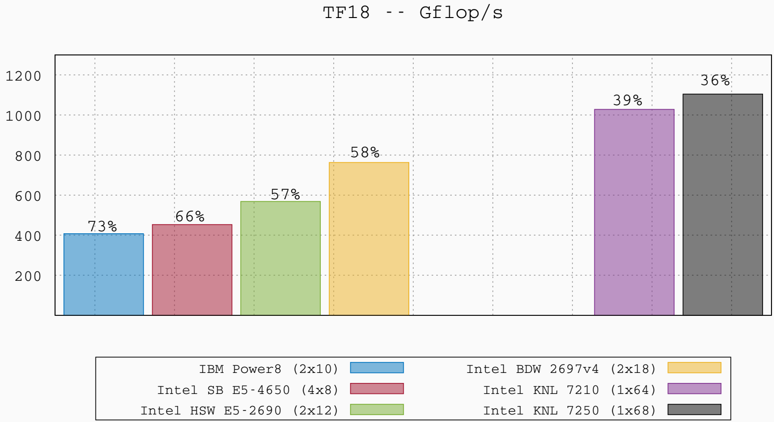
More experimental results



Credits: IBM, GENCI, IDRIS

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

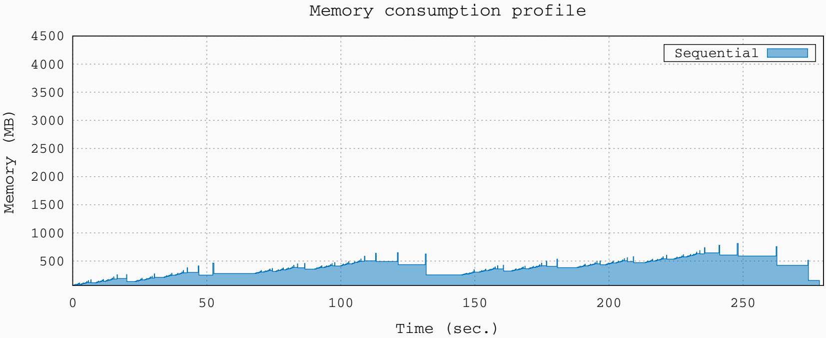
More experimental results



Credits: IBM, Intel, GENCI, CINES, IDRIS

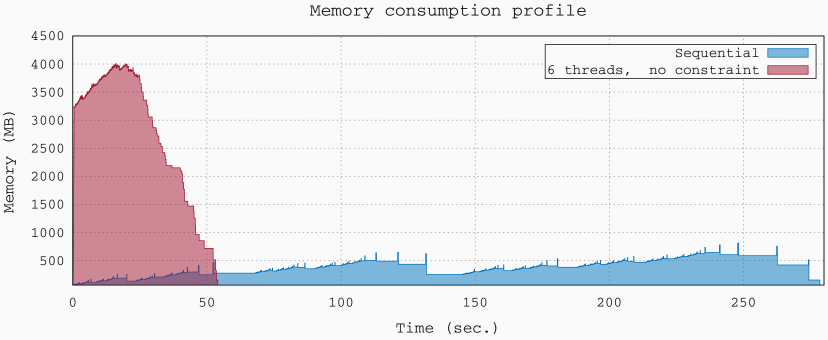
- peak is inherently unattainable: see our (and S. Kumar's and B. Bramas', and S. Nakov) work on computing meaningful performance bounds and detailed performance analysis

MEMORY-AWARE MULTIFRONTAL METHOD



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

Memory footprint in the multifrontal method

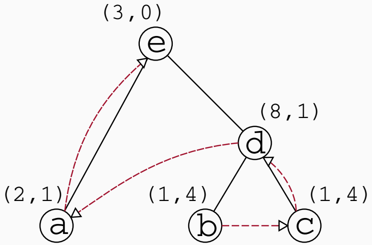


- In sequential: the memory consumption varies greatly because fronts are allocated and deallocated dynamically. The maximum memory is referred to as the sequential peak M_s.
- In parallel: the peak memory consumption M_p can be much higher because of tree parallelism.

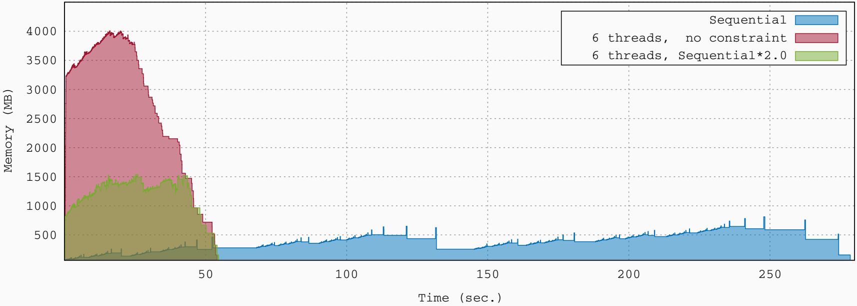
Memory-aware parallel execution

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

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



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



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

STF-BASED PARALLEL MULTIFRONTAL QR METHOD FOR HETEROGENEOUS ARCHITECTURES

GPU-based systems

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

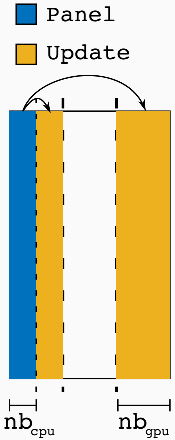
Objective

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

Issues:

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

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

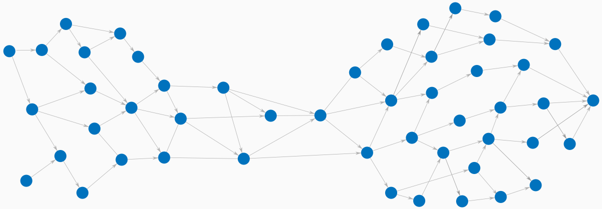


Hierarchical, dynamic partitioning

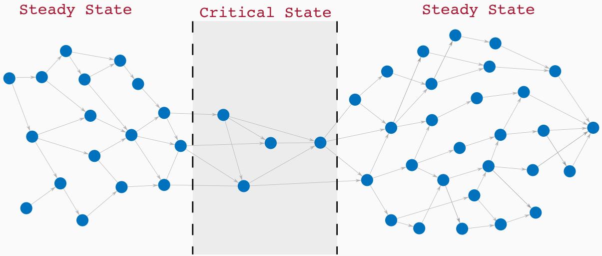
- granularity and concurrency trade-off.
- heterogeneity to be exploited.

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

DAGs are irregular and alternate rich/poor concurrency regions



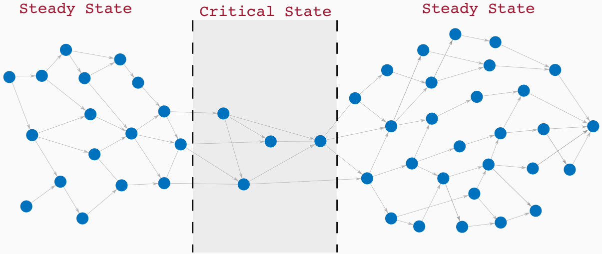
DAGs are irregular and alternate rich/poor concurrency regions



Our scheduler switches automatically between:

- Steady-state: # of ready tasks >> number of resources: execute tasks where they are best suited i.e. best acceleration factor (see HeteroPrio by Bramas et al.).
- Critical-state: # of ready tasks << number of resources: reduce the time spent on the critical path.

DAGs are irregular and alternate rich/poor concurrency regions

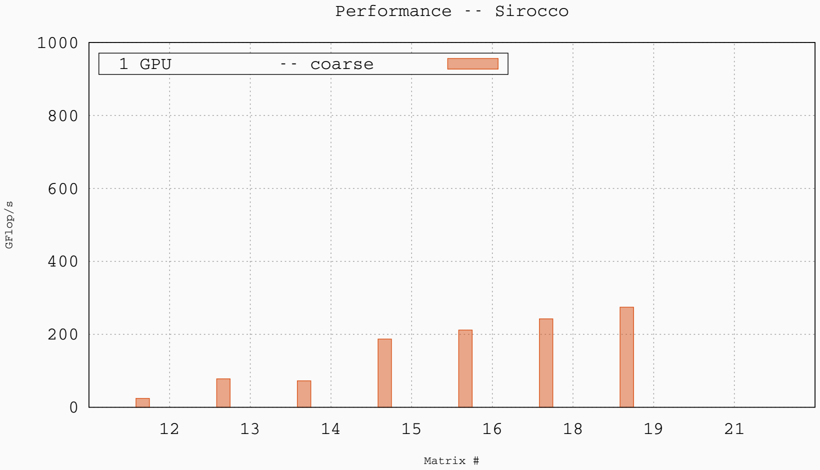


Our scheduler switches automatically between:

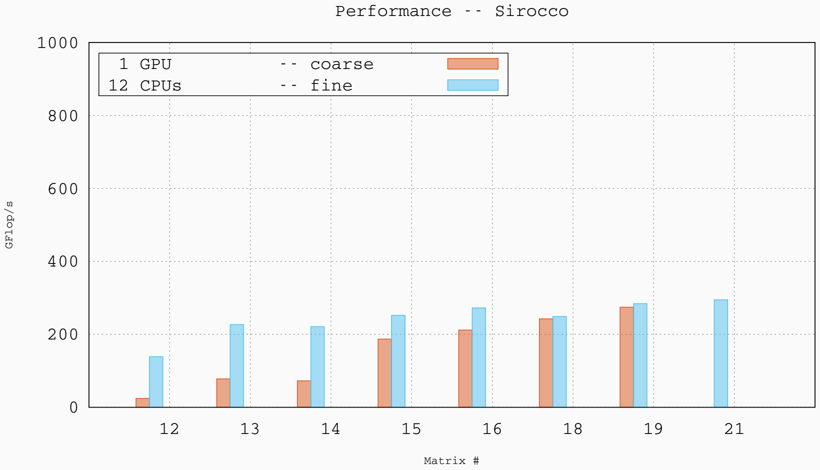
- Steady-state: # of ready tasks >> number of resources: execute tasks where they are best suited i.e. best acceleration factor (see HeteroPrio by Bramas et al.).
- Critical-state: # of ready tasks << number of resources: reduce the time spent on the critical path.

In both states prefetching is implemented to reduce the overhead of CPU-GPU communications.

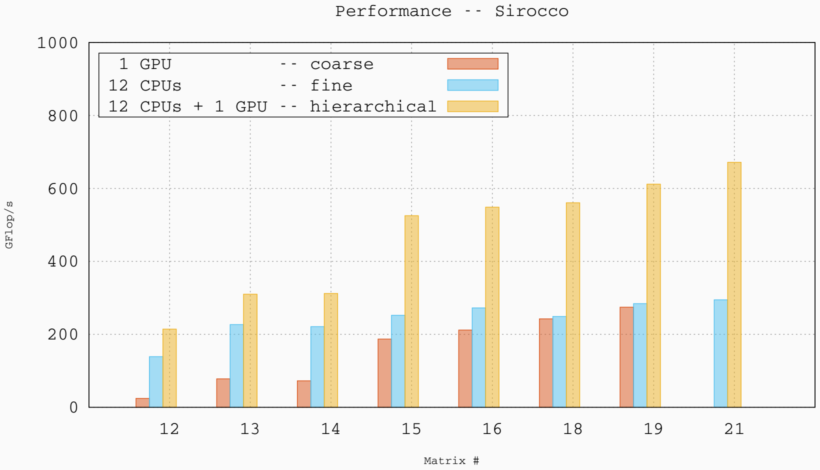
Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores + Nvidia K40 GPU



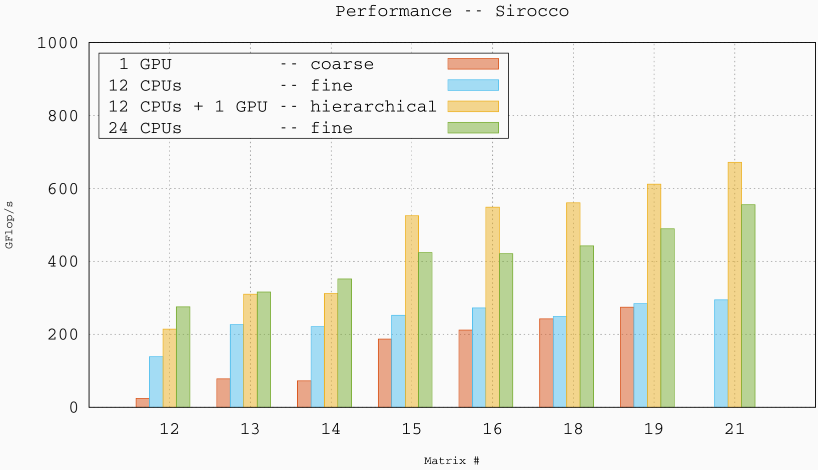
Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores + Nvidia K40 GPU



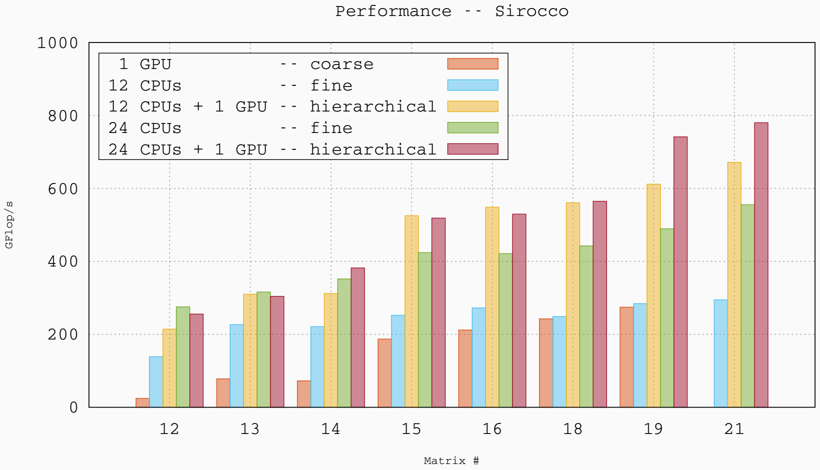
Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores + Nvidia K40 GPU



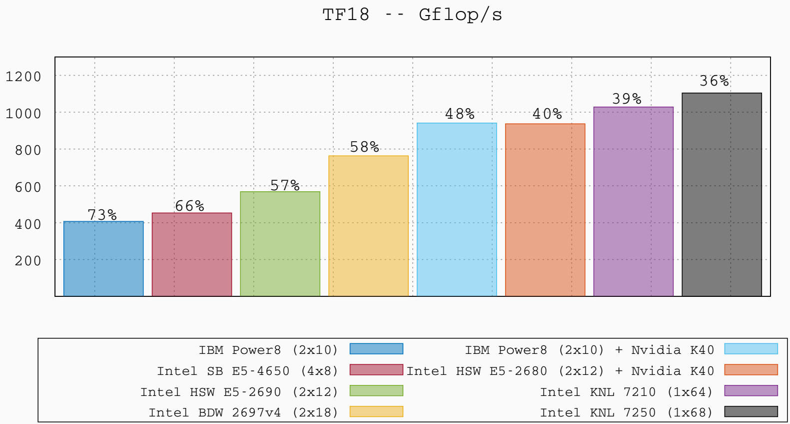
Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores + Nvidia K40 GPU



Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores + Nvidia K40 GPU



More experimental results



Credits: IBM, Intel, GENCI, CINES, IDRIS

OTHER FEATURES

- Accurate and fast simulation through the StarPU+Simgrid engine (see work by Stanisic *et al.*)
- Definition of meaningful performance bounds and detailed and accurate performance profiling (see our work and S. Kumar's and B. Bramas' and S. Nakov's)
- The asynchronous execution model allows for easy
  - Concurrent execution of different operations on different data
  - Pipelined execution of different operations on the same data

COMMERCIALS

SOLvers for Heterogeneous Architectures using Runtimes (ANR-13-MONU0007)

- Solvers (`qr_mumps`, PaStiX, Chameleon,...)
- Runtimes (StarPU)
- Scheduling
- Performance analysis

More at `http://solhar.gforge.inria.fr`

Get qr_mumps at

http://buttari.perso.enseeiht.fr/qr_mumps

or install it using Spack

Spack

```
git clone https://github.com/fpruvost/spack.git
cd spack
git checkout morse
spack install qr_mumps
```

CONCLUSIONS AND FUTURE WORK

Our experience:

- Modern runtime systems work great for implementing complex applications on single-node, accelerated systems.
- Modern runtime systems can handle very efficiently complex, heterogeneous workloads on heterogeneous architectures.
- Task-based programming models ease the development of complex features and allow the programmer to focus more on algorithms and methods than on how to implement them.

Task-based programming models and runtime systems fit all the applications and methods? Still a research subject but we're moving forward...

Future work in `qr_mumps`

- Multi-GPU: currently possible but inefficient. Must develop dedicated scheduling and mapping methods.
- Distributed-memory parallelism: data distribution and locality must be addressed.
- Pivoting: the DAG varies dynamically at run time and thus tasks submission must be controlled.

[1] E. Agullo, G. Bosilca, A. Buttari, A. Guermouche, and F. Lopez. "Exploiting a Parametrized Task Graph model for the parallelization of a sparse direct multifrontal solver." In: *Euro-Par 2016: Parallel Processing Workshops*. To appear. 2016.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems". In: *ACM Transactions On Mathematical Software* (2016). To appear.

[3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Multifrontal QR Factorization for Multicore Architectures over Runtime Systems". In: *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 521–532. ISBN: 978-3-642-40046-9. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_53.

[4] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Task-Based Multifrontal QR Solver for GPU-Accelerated Multicore Architectures." In: *HiPC*. IEEE Computer Society, 2015, pp. 54–63. ISBN: 978-1-4673-8488-9.

[5] E. Agullo et al. *Matrices Over Runtime Systems at Exascale*. Poster at the SuperComputing 2015 conference. 2015.

[6] A. Buttari. "Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C323–C345. eprint: http://epubs.siam.org/doi/pdf/10.1137/110846427. URL: http://epubs.siam.org/doi/abs/10.1137/110846427.

[7] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers". In: *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. Dec. 2015, pp. 481–490. DOI: `10.1109/ICPADS.2015.67`.

Thanks!

Questions?