TASK-BASED MULTIFRONTAL QR SOLVER FOR HETEROGENEOUS ARCHITECTURES

Florent Lopez

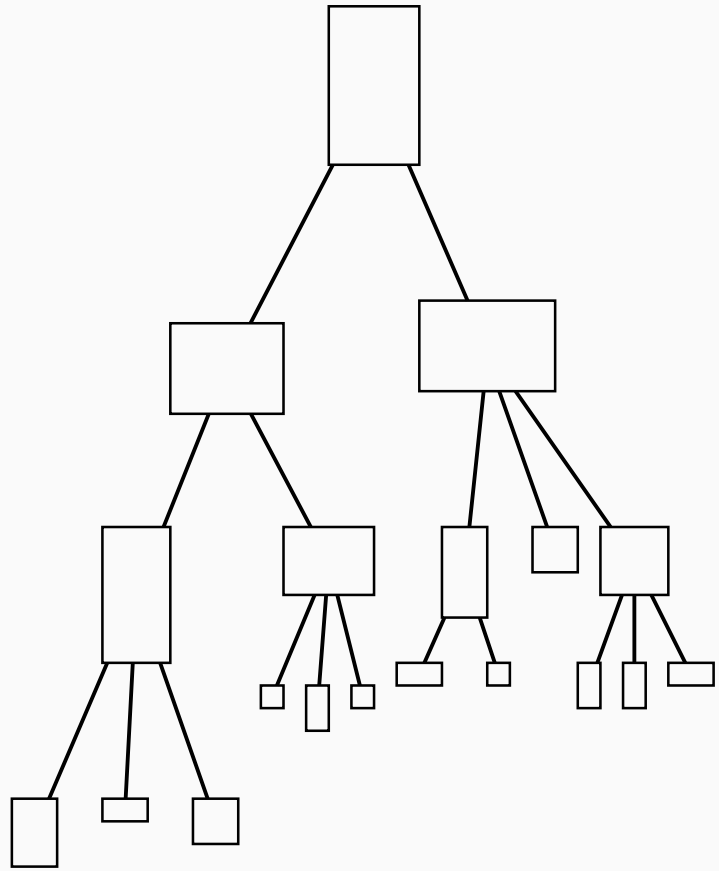
PhD days, 2015

Université Paul Sabatier-IRIT

INTRODUCTION

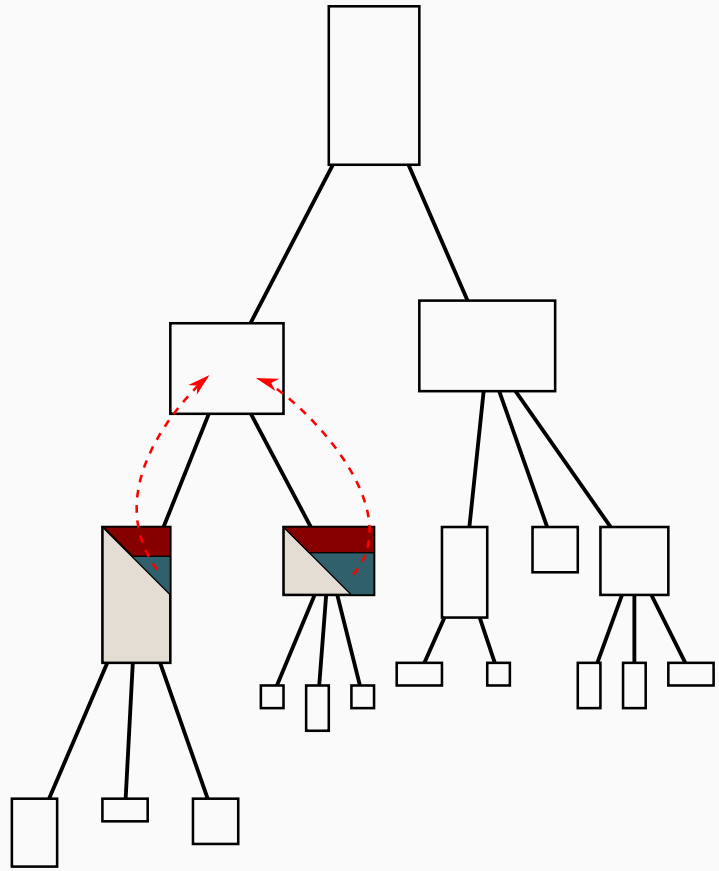
The multifrontal QR factorization is guided by a graph called *elimination tree*:

- 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.



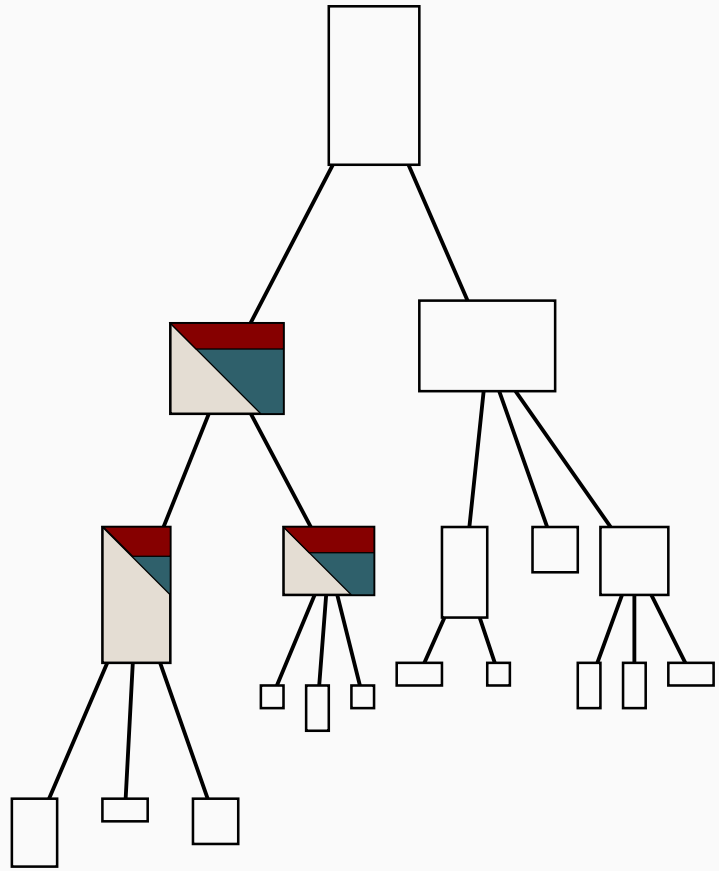
The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- 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.

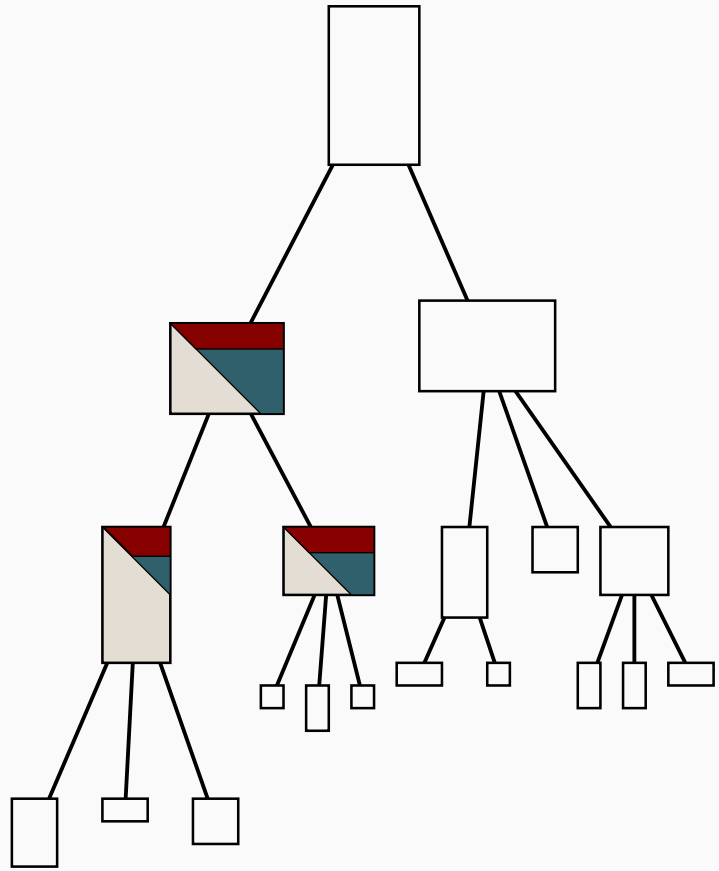


The tree is traversed in topological order (i.e., bottom-up) and, at each node, two operations are performed:

- 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.

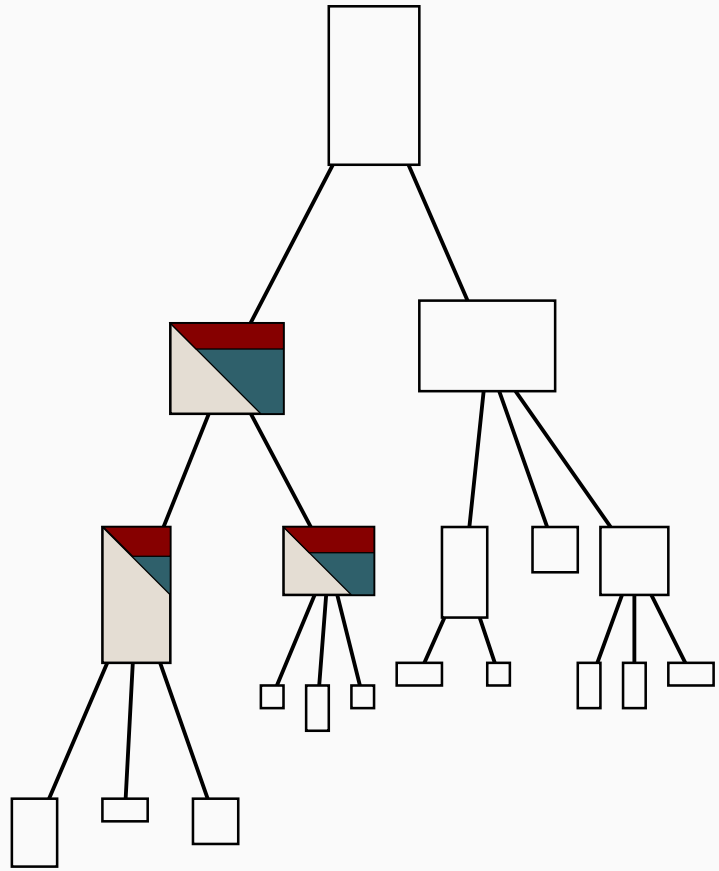


Typically two sources of parallelism are exploited in the multifrontal method



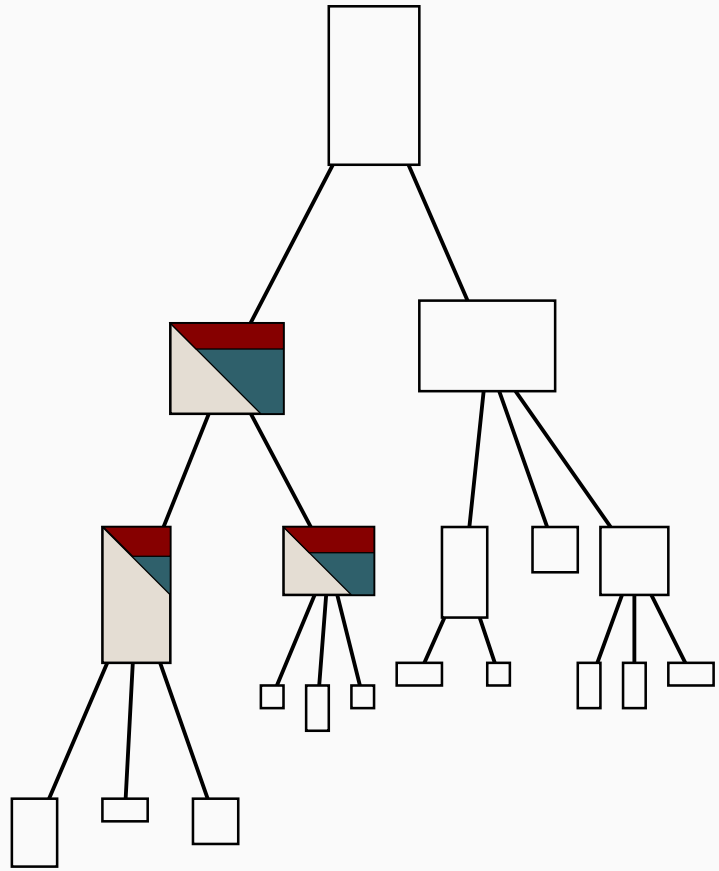
Typically two sources of parallelism are exploited in the multifrontal method

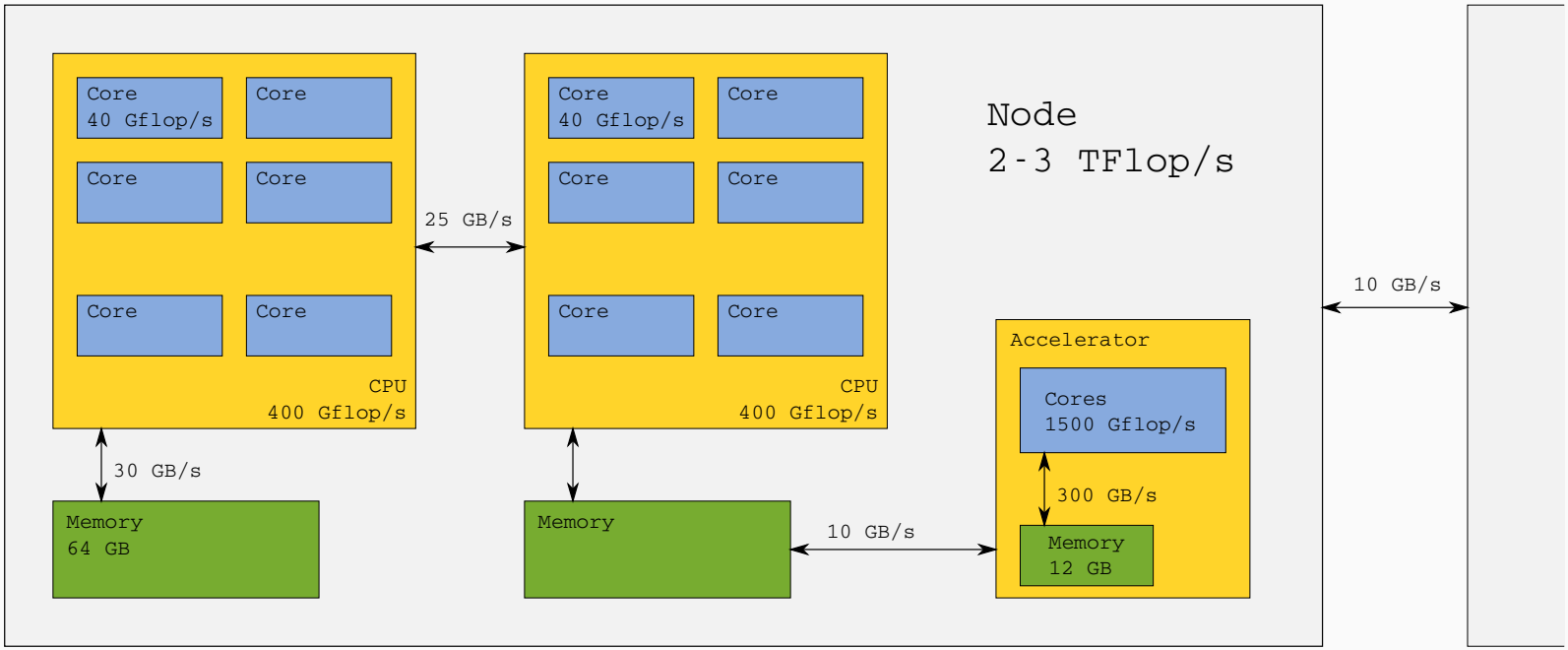
- tree-level parallelism: frontal matrices located in independent branches in the tree can be processed in parallel.



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.



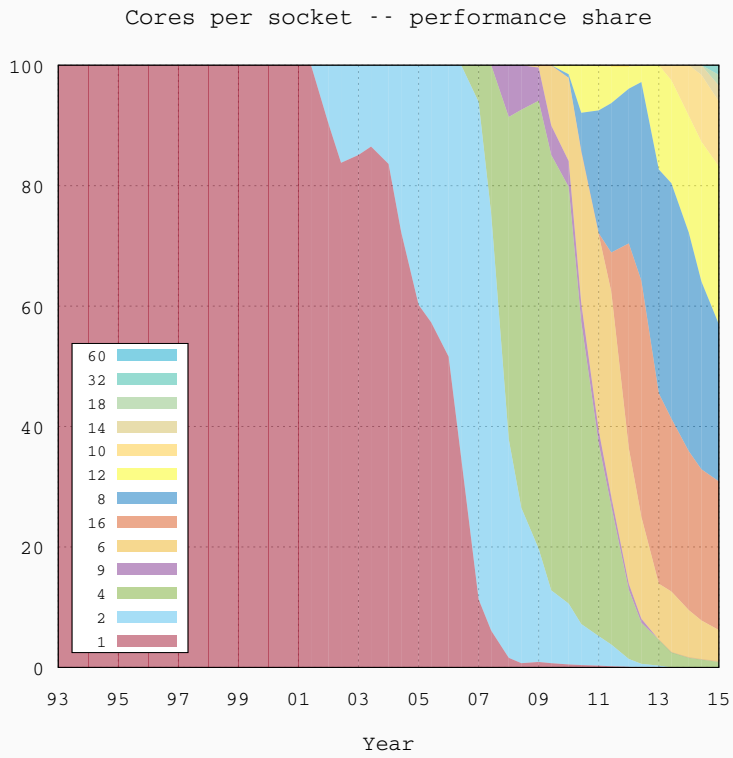


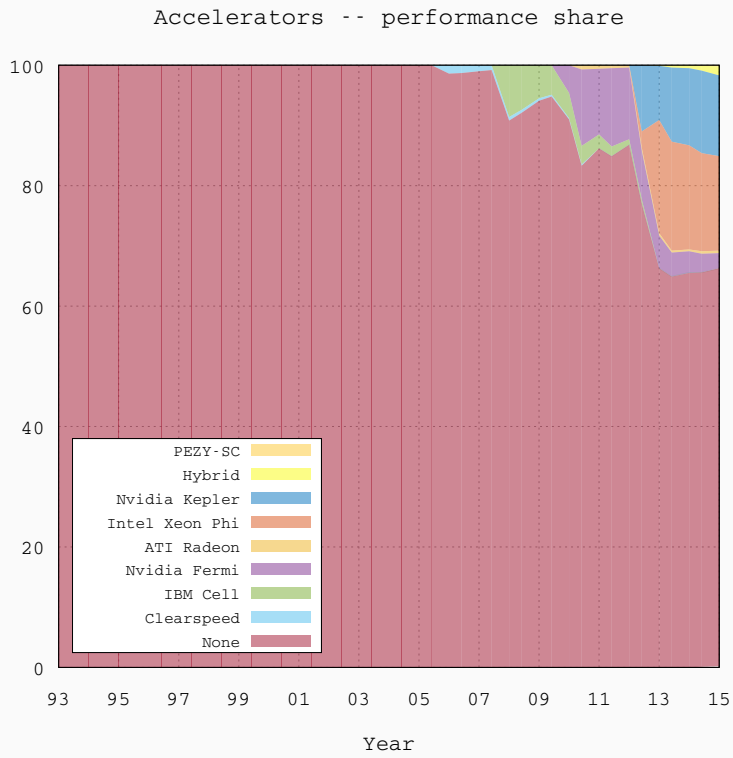
Typical HPC computing platforms are equipped with multiple nodes, connected through a high speed network, with at each nodes:

- multicore processors connected to a NUMA memory node.
- one or more accelerators.

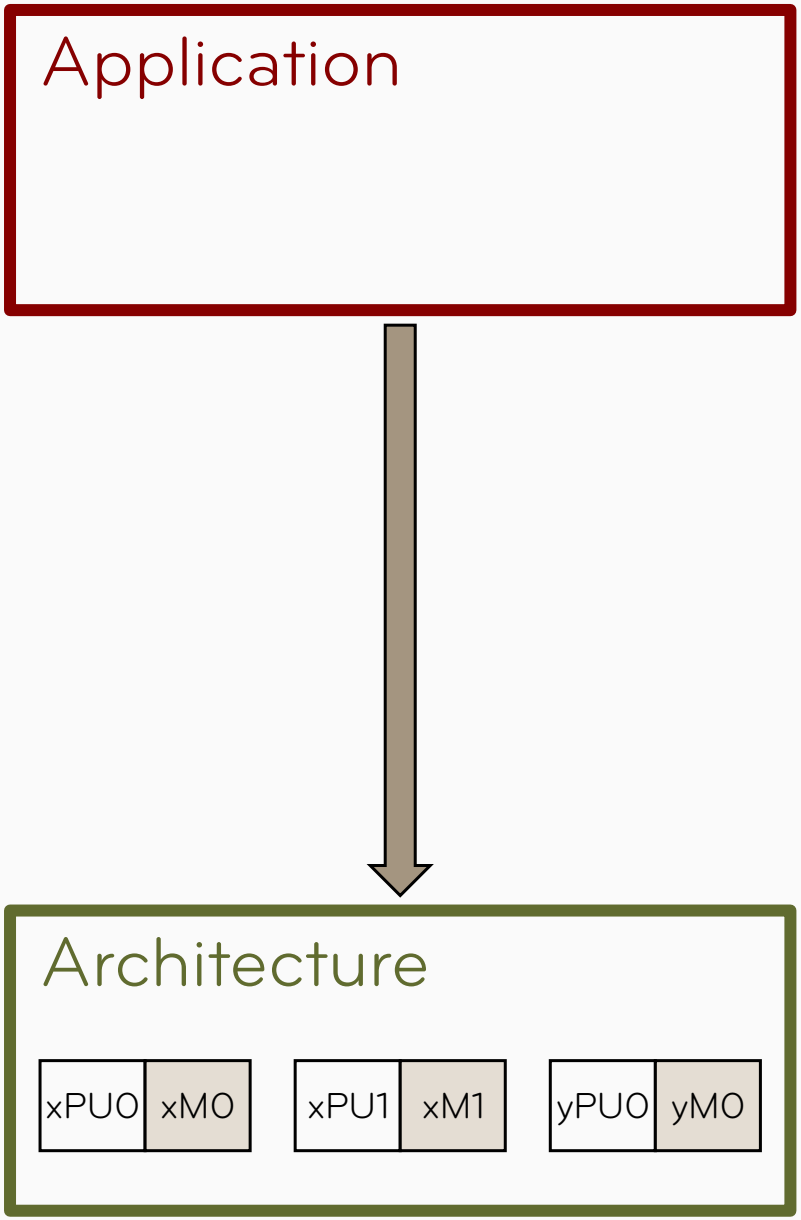
Current HPC platform are extremely heterogeneous with diverse resource capabilities and memory speeds offered by these

ARCHITECTURES

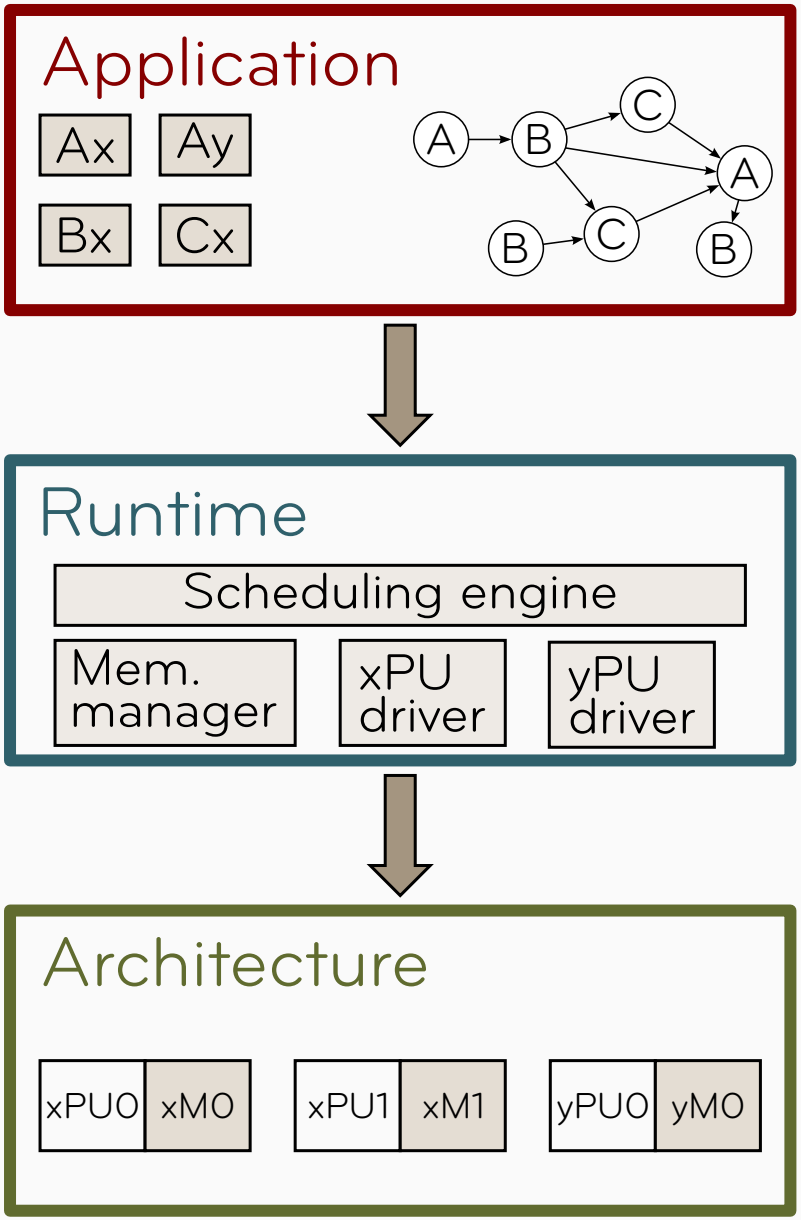




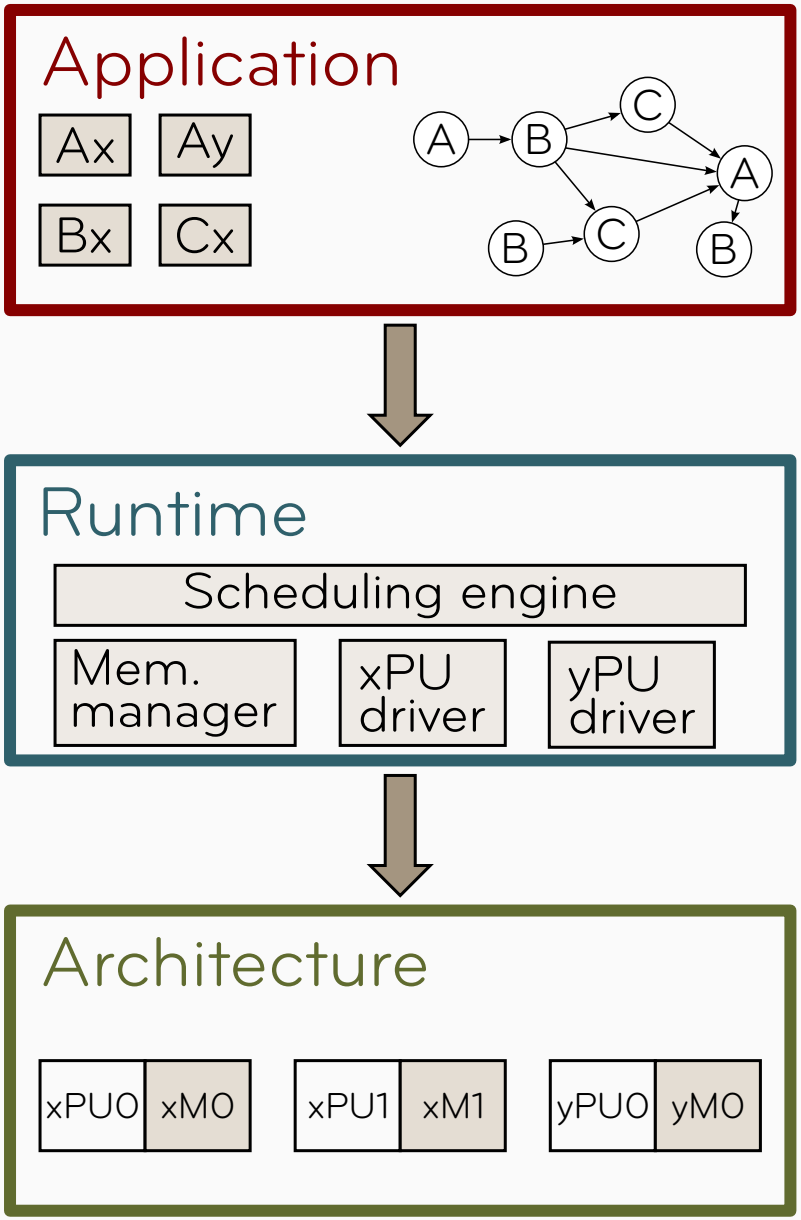
According to the Top500 lists, multicore architectures introduced in 2001 nowadays represents 100 % of the performance share in the list. Accelerator, such as GPU and Xeon Phi devices, started gaining interests in the HPC community in 2005 and became very popular ever since.



- 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 of tasks.

THE SEQUENTIAL TASK FLOW MODEL

In the Sequential Task Flow (STF) model:

- The parallel code looks exactly the same as the sequential one except that operations are not executed but submitted to the system in the form of tasks
- Depending on how tasks access data and on the (sequential) order of submission, the runtime infers dependencies among them and builds a DAG
- The runtime scheduler deploys the DAG on the underlying architecture
- The runtime memory manager moves data from one memory to another and maintains the global memory coherency
- Equivalent to Superscalar techniques in microprocessors

Runtimes relying on STF: StarPU, QUARK, SMPss, OpenMP 4.0

THE STF 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 STF 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);
```

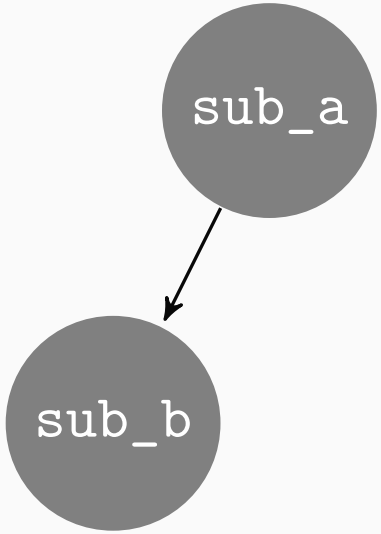
THE STF MODEL: A SIMPLE EXAMPLE

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);
```



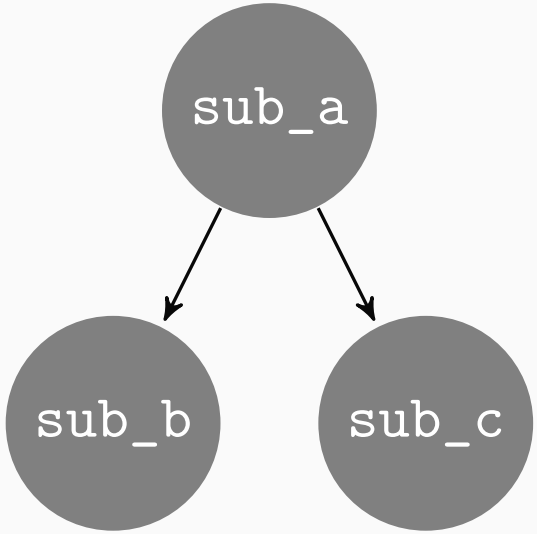
THE STF MODEL: A SIMPLE EXAMPLE

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);
```



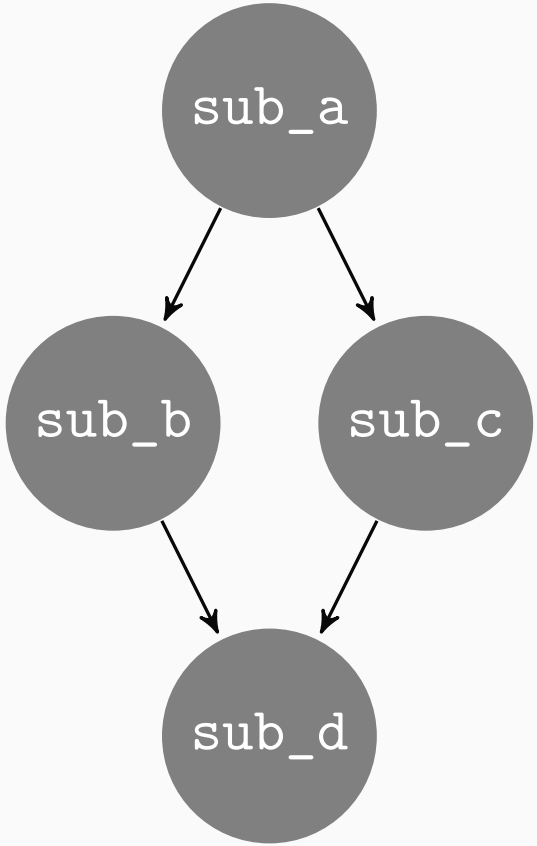
THE STF MODEL: A SIMPLE EXAMPLE

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);
```



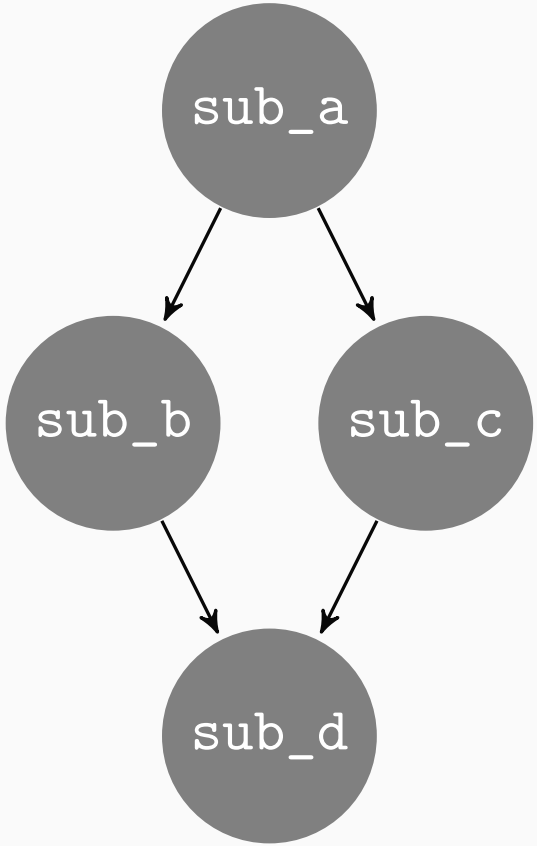
THE STF MODEL: A SIMPLE EXAMPLE

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.

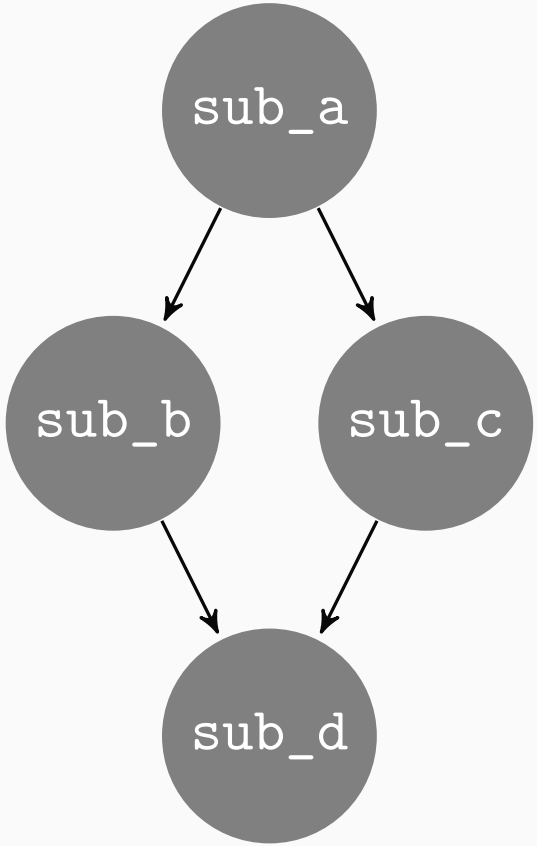
THE STF MODEL: A SIMPLE EXAMPLE

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( );
```



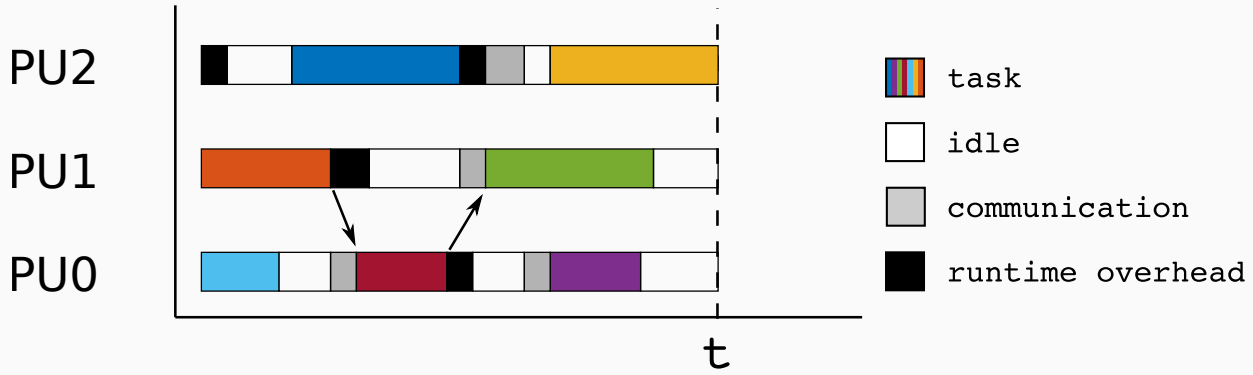
sub_b and **sub_c** can be executed in parallel. If **sub_a** is executed on CPU and **sub_b** on GPU, **x** will be automatically transferred.

PERFORMANCE ANALYSIS APPROACH

The parallel efficiency can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on execution time on p resources corresponding to the best schedule under the following assumptions:

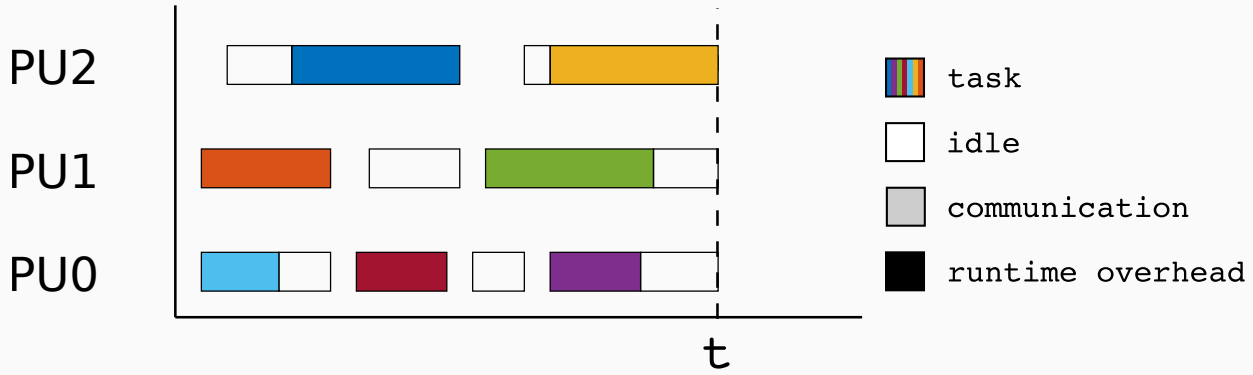


The parallel efficiency can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on execution time on p resources corresponding to the best schedule under the following assumptions:

- No communications and no runtime overhead

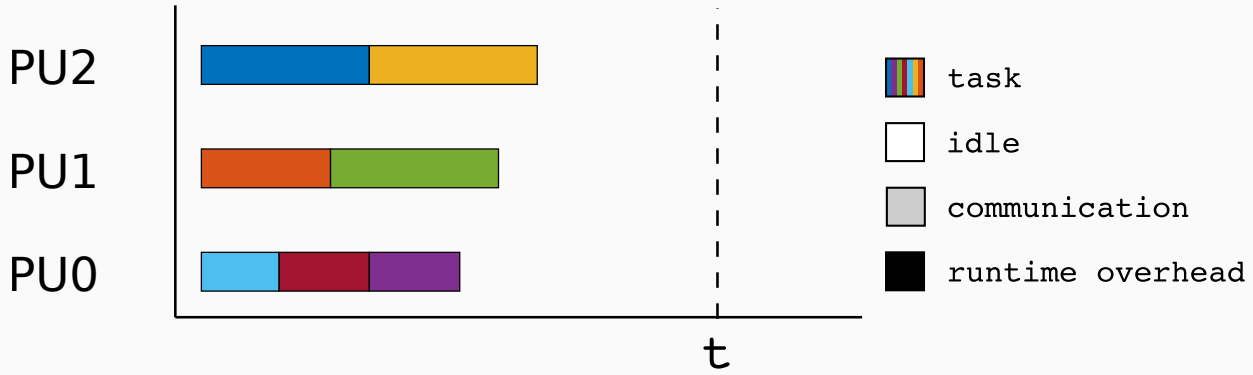


The parallel efficiency can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on execution time on p resources corresponding to the best schedule under the following assumptions:

- No communications and no runtime overhead
- No dependencies between tasks (embarrassing parallelism)

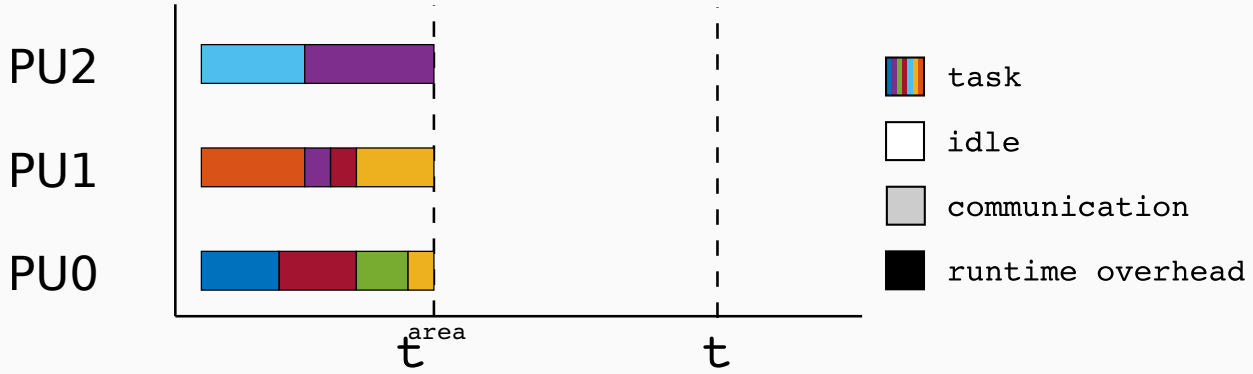


The parallel efficiency can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on execution time on p resources corresponding to the best schedule under the following assumptions:

- No communications and no runtime overhead
- No dependencies between tasks (embarrassing parallelism)
- Tasks are moldable



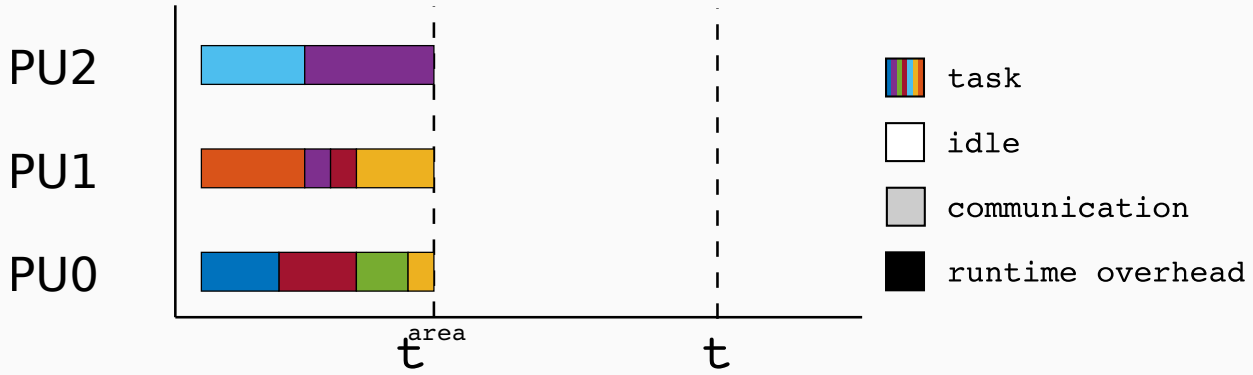
Because of our assumption we call this reference time $t^{area}(p)$

The parallel efficiency can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on execution time on p resources corresponding to the best schedule under the following assumptions:

- No communications and no runtime overhead
- No dependencies between tasks (embarrassing parallelism)
- Tasks are moldable



We consider $t^{area}(p)$ when there is no performance loss resulting from data partitioning denoted $\tilde{t}^{area}(p)$

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}^{area}_t(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}^{area}_t(p)}{t^{area}_t(p)}}^{e_g} \cdot \overbrace{\frac{t^{area}_t(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

e_g: the granularity efficiency. Measures the impact of data partitioning and use of parallel algorithms

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

e_t: the Task efficiency. Measures how well the assignment of tasks to processing units matches the tasks properties to the units capabilities

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}^{area}_t(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}^{area}_t(p)}{t^{area}_t(p)}}^{e_g} \cdot \overbrace{\frac{t^{area}_t(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

e_r: the runtime efficiency. Measures how the runtime overhead affects performance

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

e_c: the communication efficiency. measures the cost of communications with respect to the actual work done due to data transfers between workers.

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

e_p: the pipeline efficiency. Measures how much concurrency is available and how well it is exploited.

The execution time $t(p)$ can be decomposed in the following three terms:

- $t_t(p)$: the time spent executing tasks
- $t_r(p)$: the overhead of the runtime system
- $t_c(p)$: The time spent performing communications.
- $t_i(p)$: idle time

The overall efficiency can thus be written as:

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$

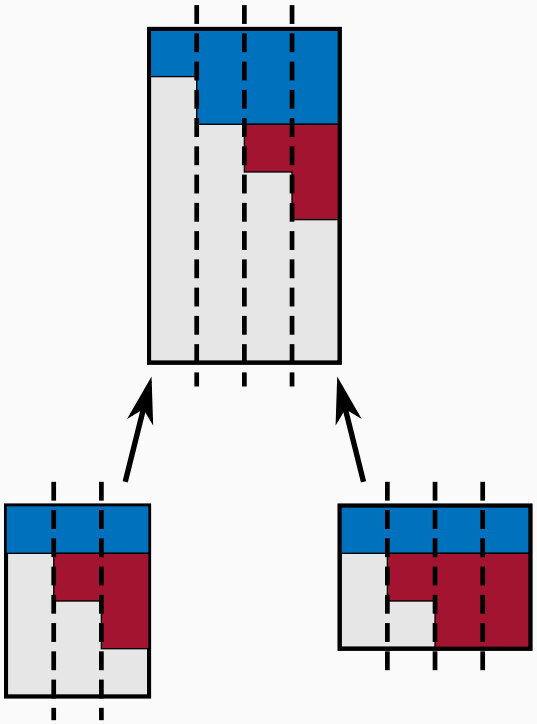
$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

with:

$e_t \cdot e_p$ measures the quality of the scheduling and is ≤ 1

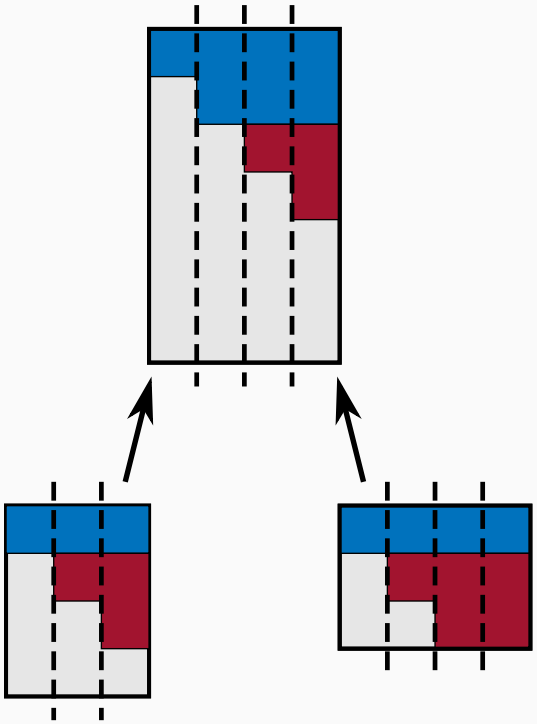
STF MULTIFRONTAL QR METHOD FOR MULTICORE ARCHITECTURES

Sequential qr_mumps code



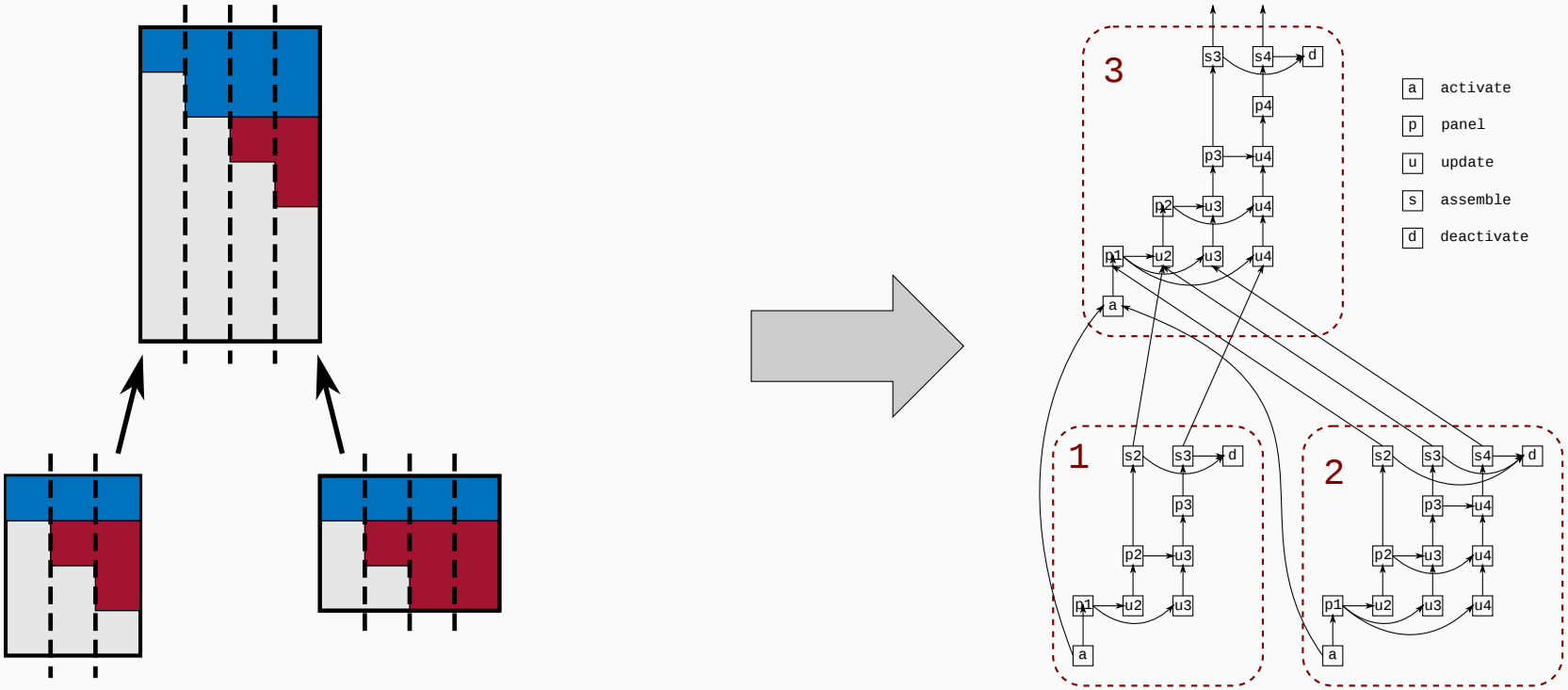
```
do f=1, nfronts ! in postorder
   ! activate front
   call activate(f)
   ! init front
   call init(f)

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

STF parallel qr_mumps code



```
do f=1, nfronts ! in postorder
   ! compute structure and register handles
   call activate(f)
   ! allocate and initialize front
   call starpu_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 starpu_submit(assemble, c(j):R, f:RW)
      end do
      ! Deactivate child
      call starpu_submit(deactivate, c:RW)
   end do

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

The tree is transformed into a DAG



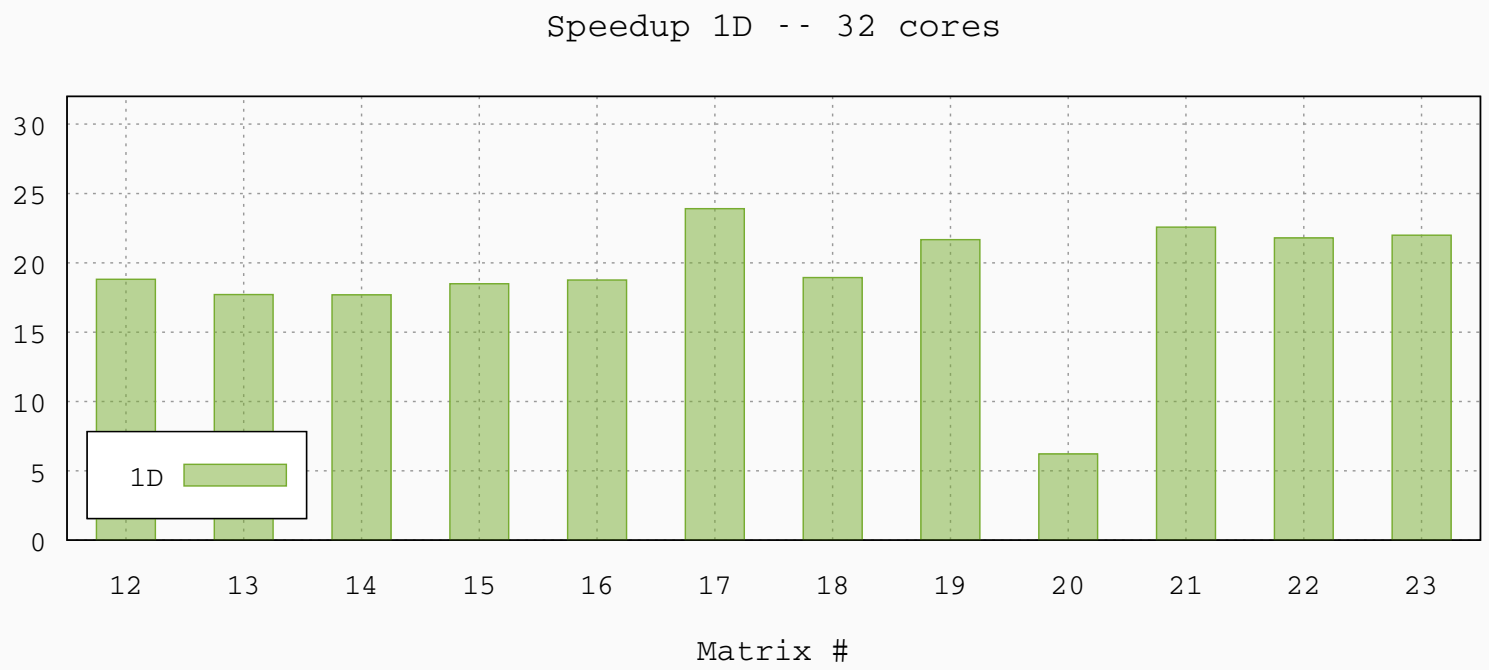
- Seamless exploitation of tree and node parallelism.
- Inter-level concurrency (father-child pipelining).

#	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

System Ada:

- IBM x3750-M4
- Intel Sandy Bridge E5-4650 @ 2.7 GHz, 4×8 cores
- 128 GB memory (NUMA)

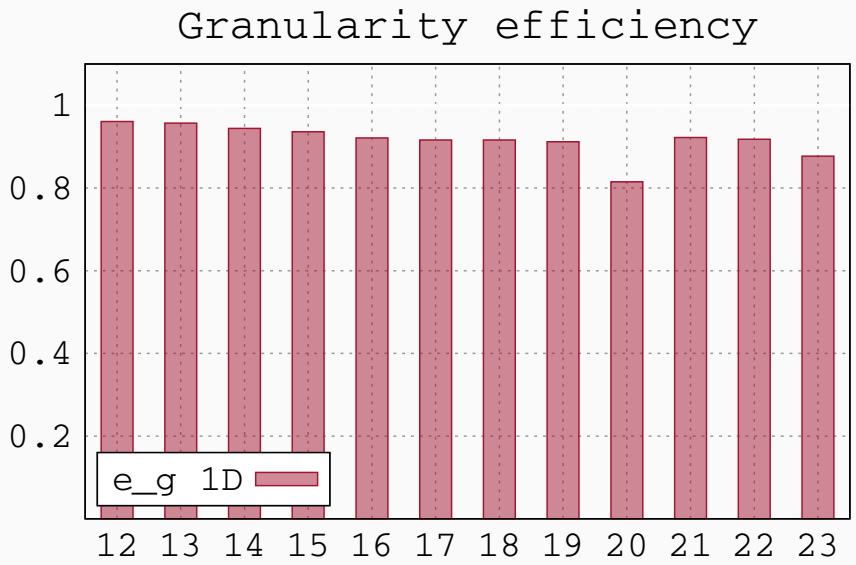
EXPERIMENTAL RESULTS: SPEEDUPS

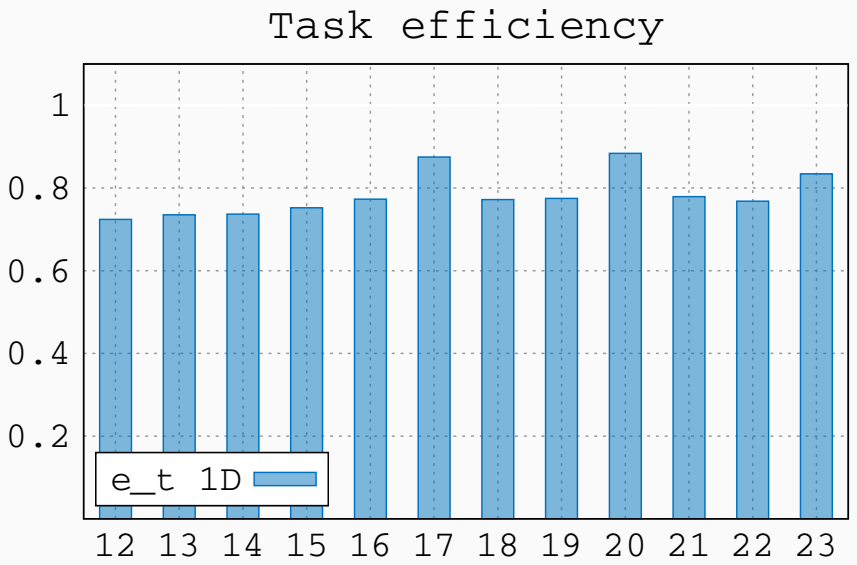


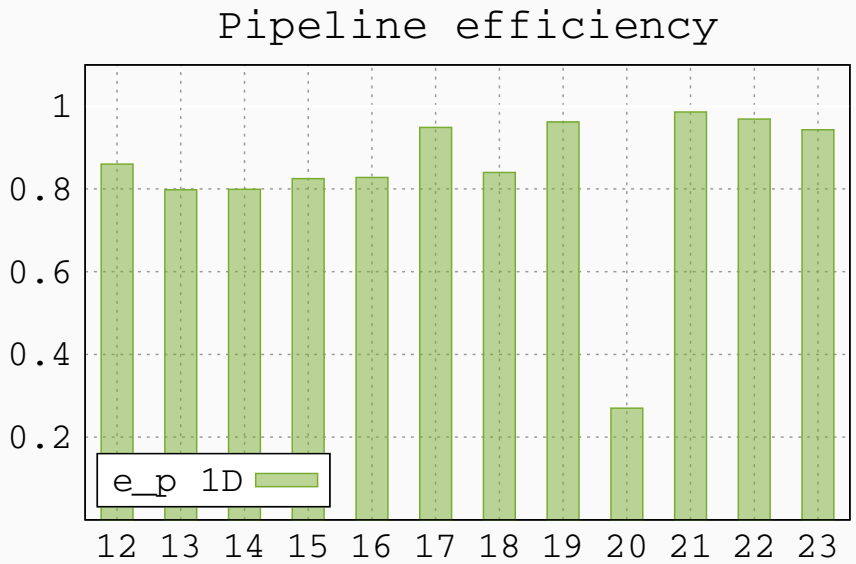
The task-based multifrontal method, implemented with a STF parallel model on top of StarPU offers good speedups on 32 cores:

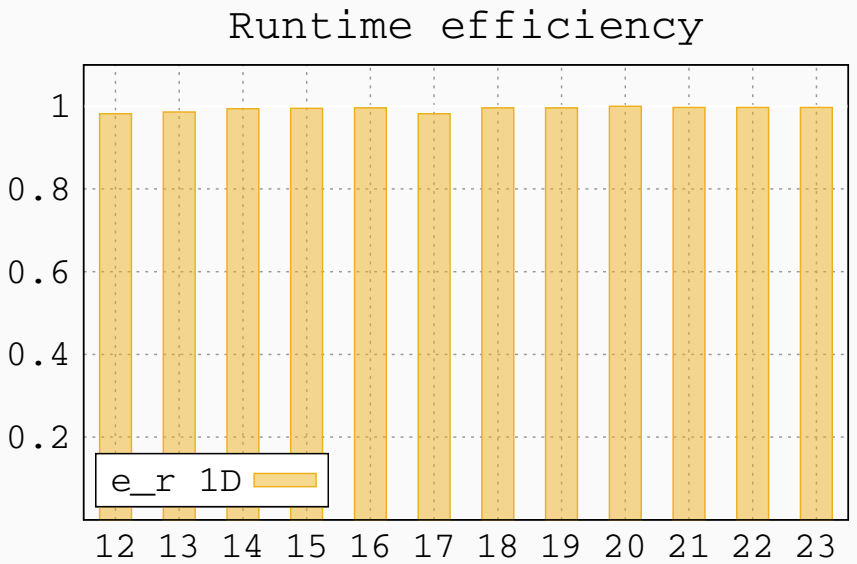
- speedup increases with problem size with very low speedup for some problem such as matrix # 20
- we use a detailed performance analysis to determine the limiting factors of the STF 1D approach

EXPERIMENTAL RESULTS: EFFICIENCY BREAKDOWN









2D PARTITIONING + CA FRONT FACTORIZATION

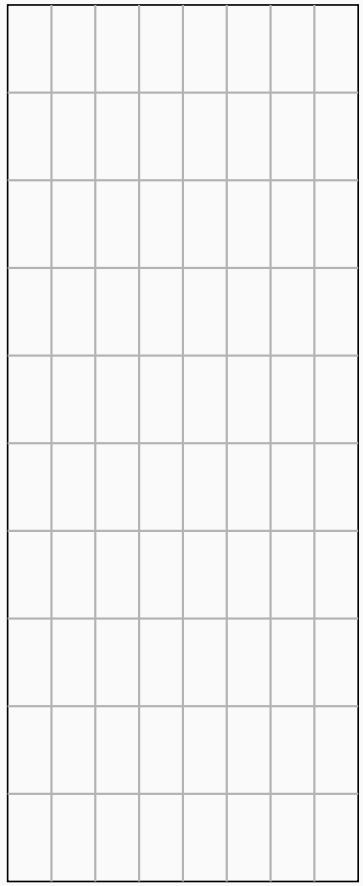
1D partitioning is not good for (strongly) overdetermined matrices:

- ▼ 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



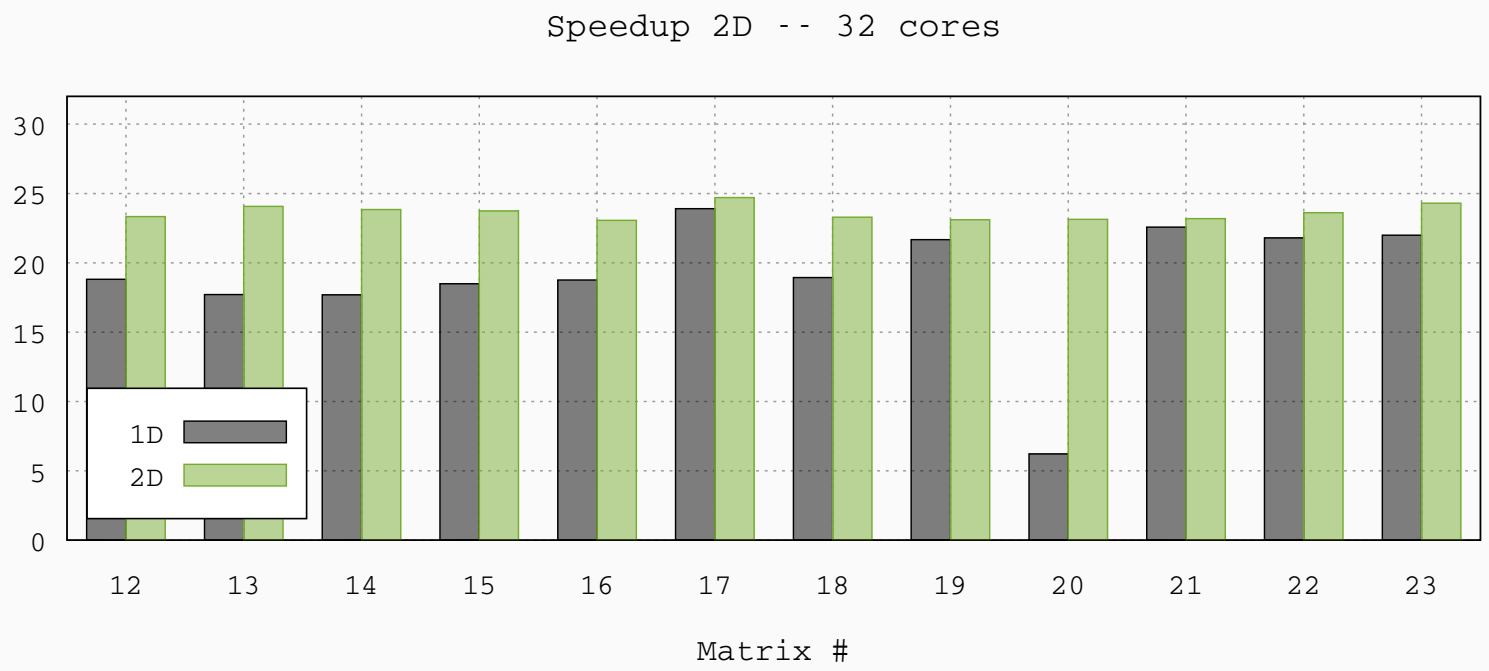
1D PARTITIONING FRONT FACTORIZATION

```
do f=1, nfronts                                    ! in postorder

   call activate(f)                                ! activate front
   call submit(init, f:RW)                         ! init front

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

   do p=1, f%n
      ! panel reduction of column p
      call submit(panel, f(p):RW)
      do u=p+1, f%n
         ! update of column u with panel p
         call submit(update, f(p):R, f(u):RW)
      end do
   end do
end do

call wait_tasks_completion()                       ! wait for the tasks to be executed
```

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
```

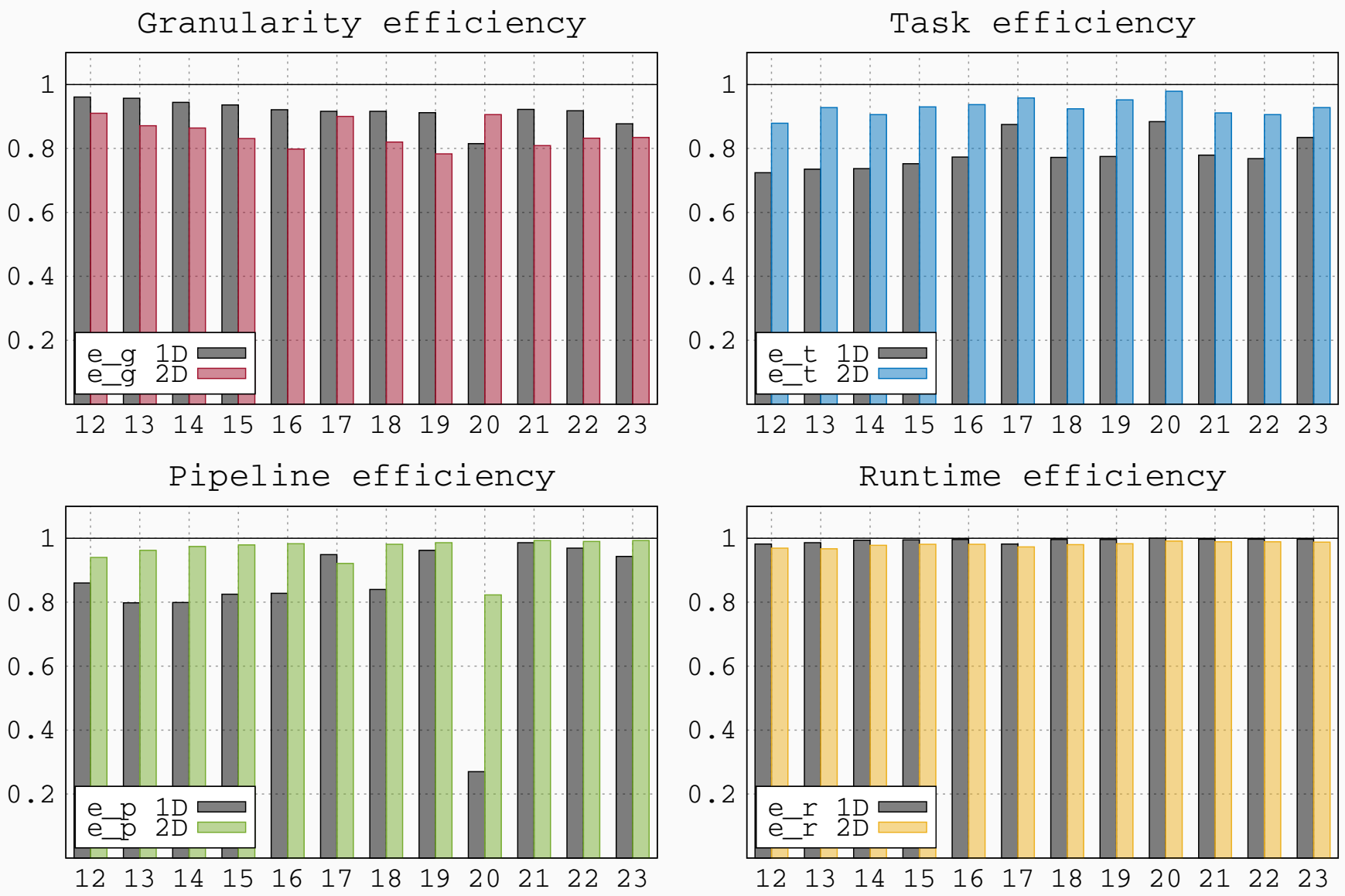
EXPERIMENTAL RESULTS: SPEEDUPS



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 on a 32 cores systems.
- We perform a comparative performance analysis wrt to the 1D case to show the benefits of the 2D scheme.

EXPERIMENTAL RESULTS: EFFICIENCY BREAKDOWN



STF MULTIFRONTAL QR METHOD FOR HETEROGENEOUS ARCHITECTURES

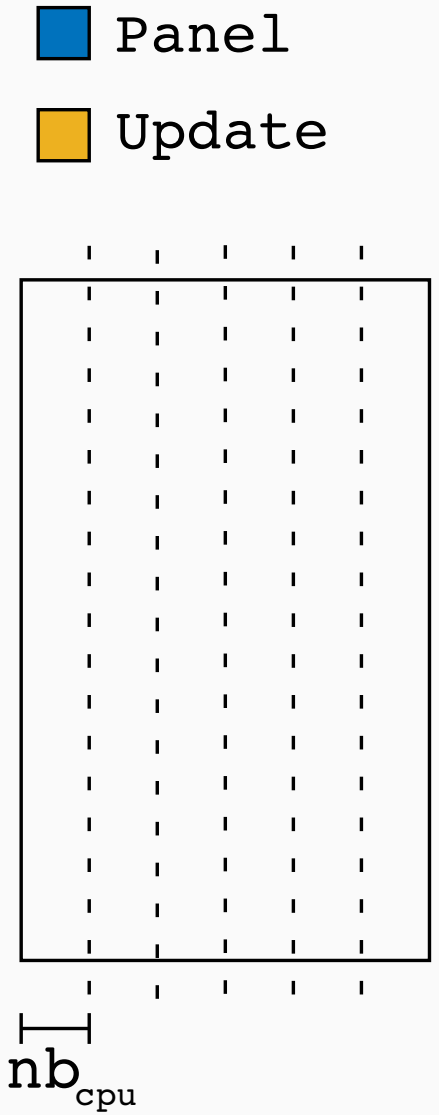
STF parallel qr_mumps code

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

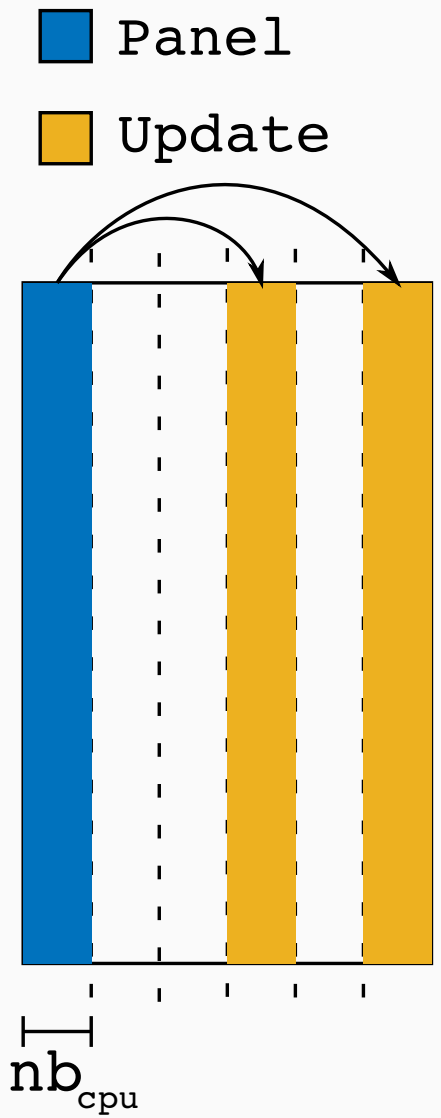
STF parallel qr_mumps code

```
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)  ← CPU and GPU task
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

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

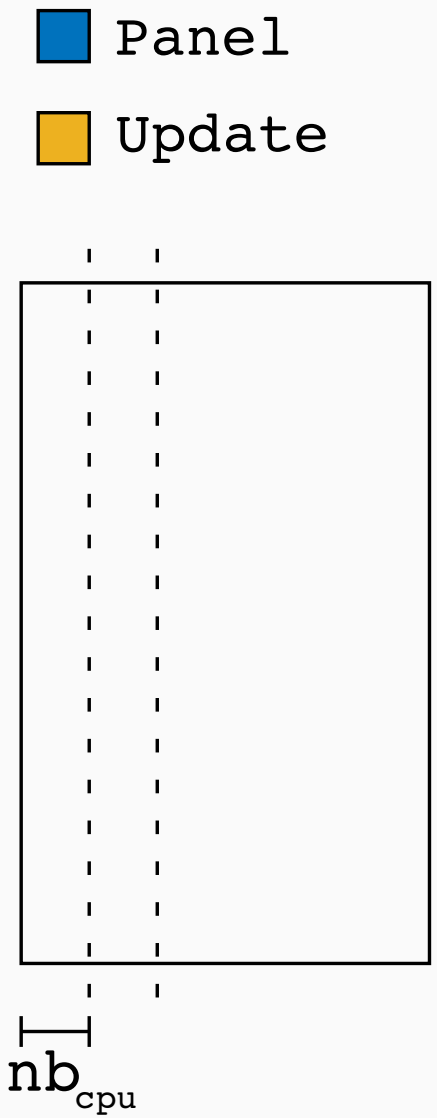
- Fine grain [3, 2, 5]



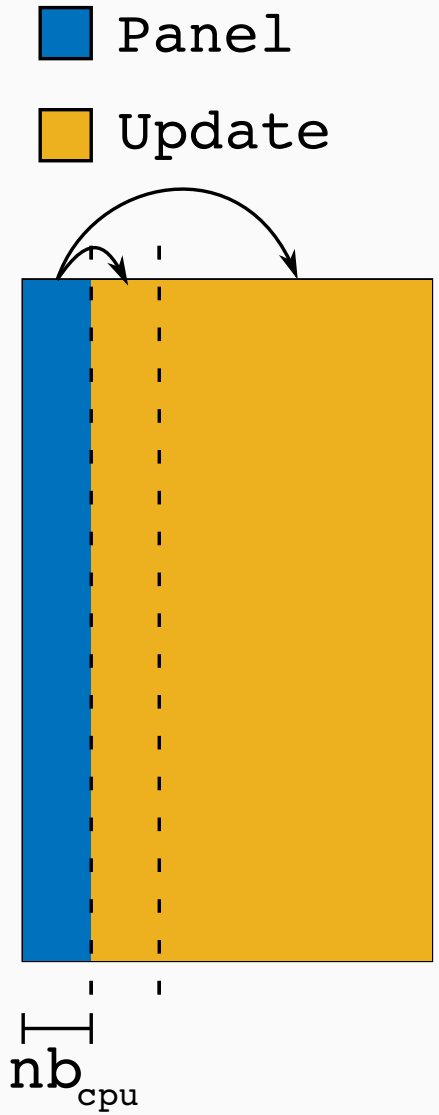
- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.



- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)

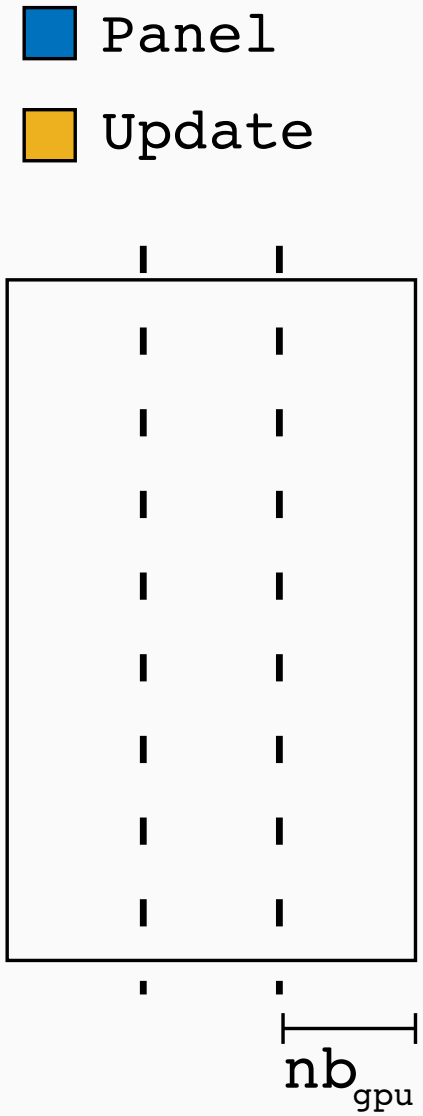


- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)
  - ▲ optimum granularity for GPU.
  - ▼ limited concurrency.



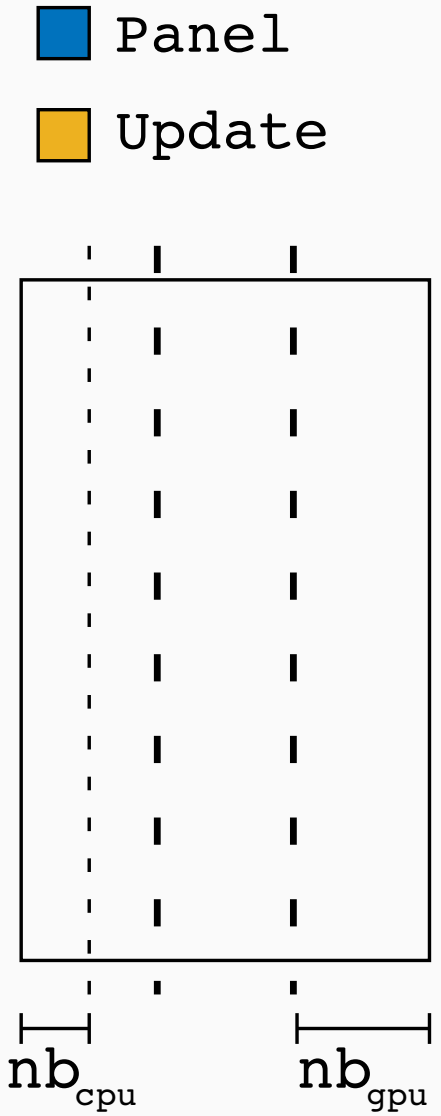
- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)
  - ▲ optimum granularity for GPU.
  - ▼ limited concurrency.

Hierarchical



- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)
  - ▲ optimum granularity for GPU.
  - ▼ limited concurrency.

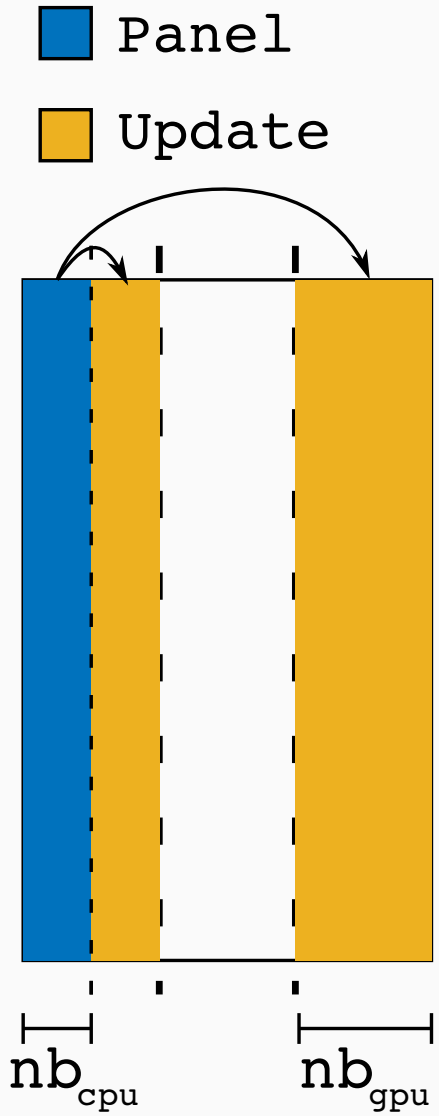
Hierarchical



- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)
  - ▲ optimum granularity for GPU.
  - ▼ limited concurrency.

Hierarchical

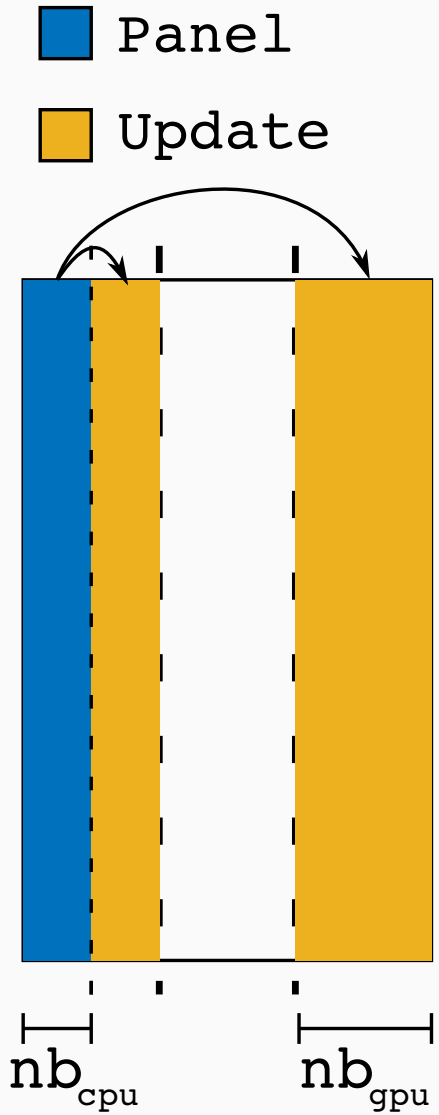
- ▲ granularity and concurrency trade-off.
- ▲ heterogeneity to be exploited.



- Fine grain [3, 2, 5]
  - ▲ high concurrency.
  - ▼ low tasks efficiency.
- Coarse grain (Magma)
  - ▲ optimum granularity for GPU.
  - ▼ limited concurrency.

Hierarchical

- ▲ granularity and concurrency trade-off.
- ▲ heterogeneity to be exploited.
- requires dynamic repartitioning.



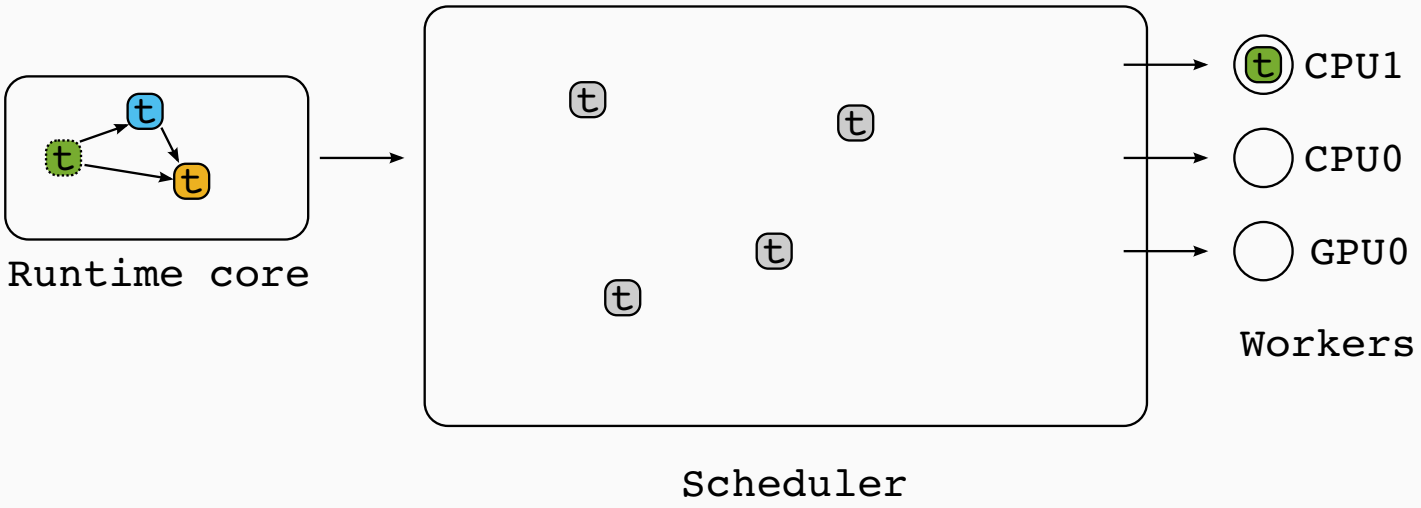
- The partitioning is done dynamically through a partitioning task.
- the partitioning is symbolic: neither allocations nor data copies.
- the runtime ensures the consistency of data.

```
forall outer panels o_p=1...o_n in f
   ! partition (outer) block column f(o_p) into
   ! i_n inner block columns f(o_p,1) .. f(o_p,i_n)
   call submit(partition, f(o_p):R, f(o_p,1):W, .., f(o_p,i_n):W)     ←−

   forall inner panels i_p=1..i_n
      ! panel reduction of inner block column i_p
      call submit(inner_panel, f(i_p):RW)

      ! update (inner) column in_u with panel p
      forall inner blockcolumns i_u=i_p+1..i_n in f(o_p)
        call submit(inner_update, f(i_p):R, f(i_u):RW)
      end do

      ! update outer block column o_u with panel i_p
      forall outer blockcolumns o_u=o_p+1..o_n
        call submit(outer_update, f(i_p):R,f(o_u):RW)
      end do
   end do

   ! unpartition (outer) block column
   call submit(unpartition, f(o_p,1):R..f(o_p,i_n):R, f(o_p):W)       ←−
end do
```

Multiple sources of heterogeneity:

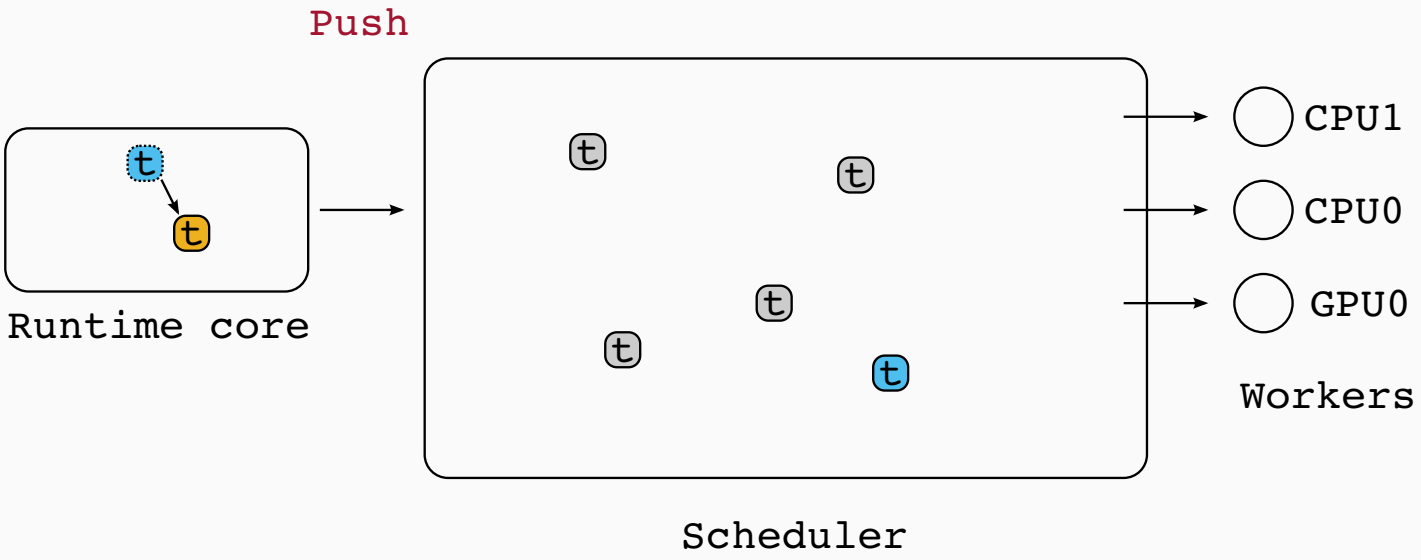
- Workload heterogeneity:
  - The hierarchical partitioning scheme produces fine and coarse granularity tasks.
  - The variability of front sizes in the etree induces a big variation of tasks granularity.
  - Compute-bound and memory-bound tasks.
- Architecture heterogeneity:
  - Processor speeds.
  - Memory speeds.
  - Memory transfers.

Efficiently execute the task graph on the architecture requires a clever scheduling strategy capable of taking into account resource capabilities.



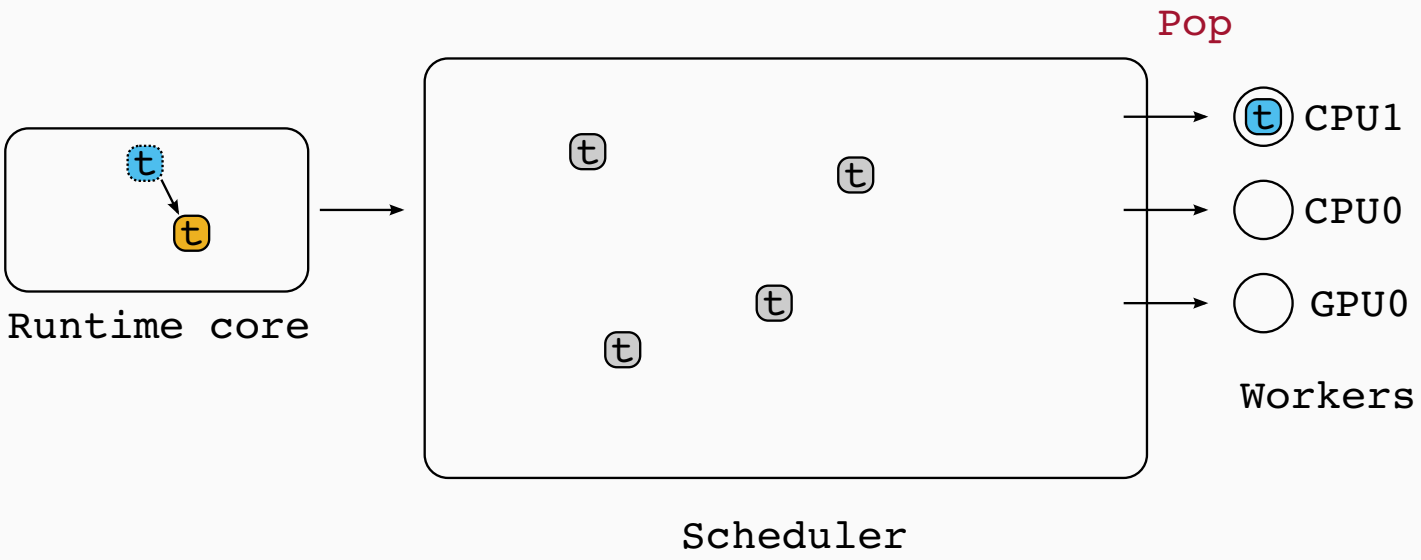
Building Blocks

- A task container storing tasks ready for execution.



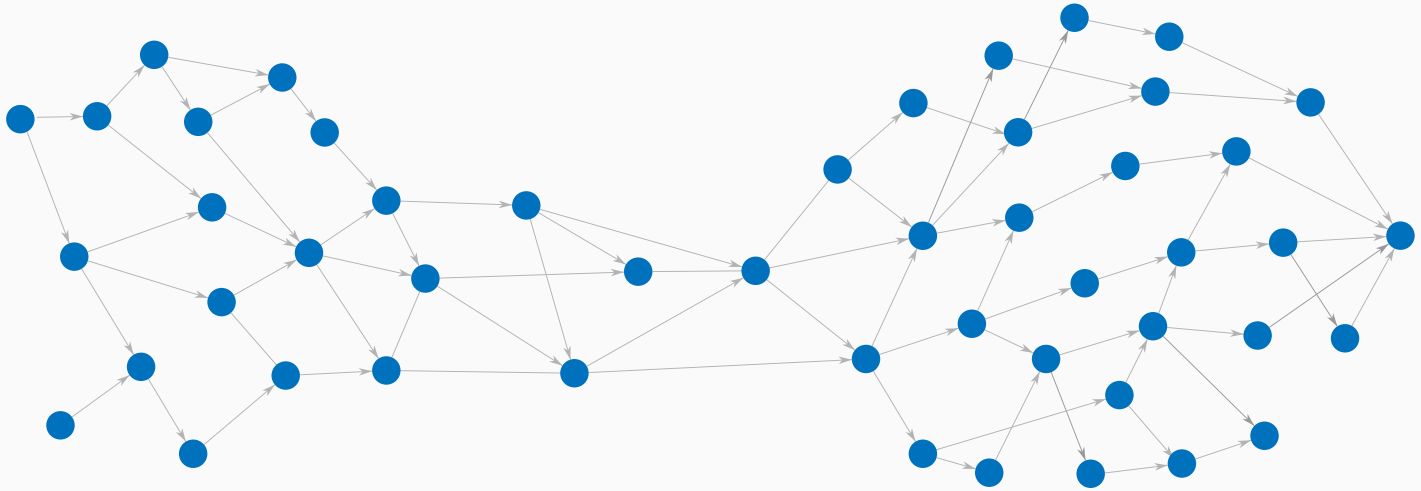
Building Blocks

- A task container storing tasks ready for execution.
- Two methods:
  - **push**: put ready tasks in the scheduler.

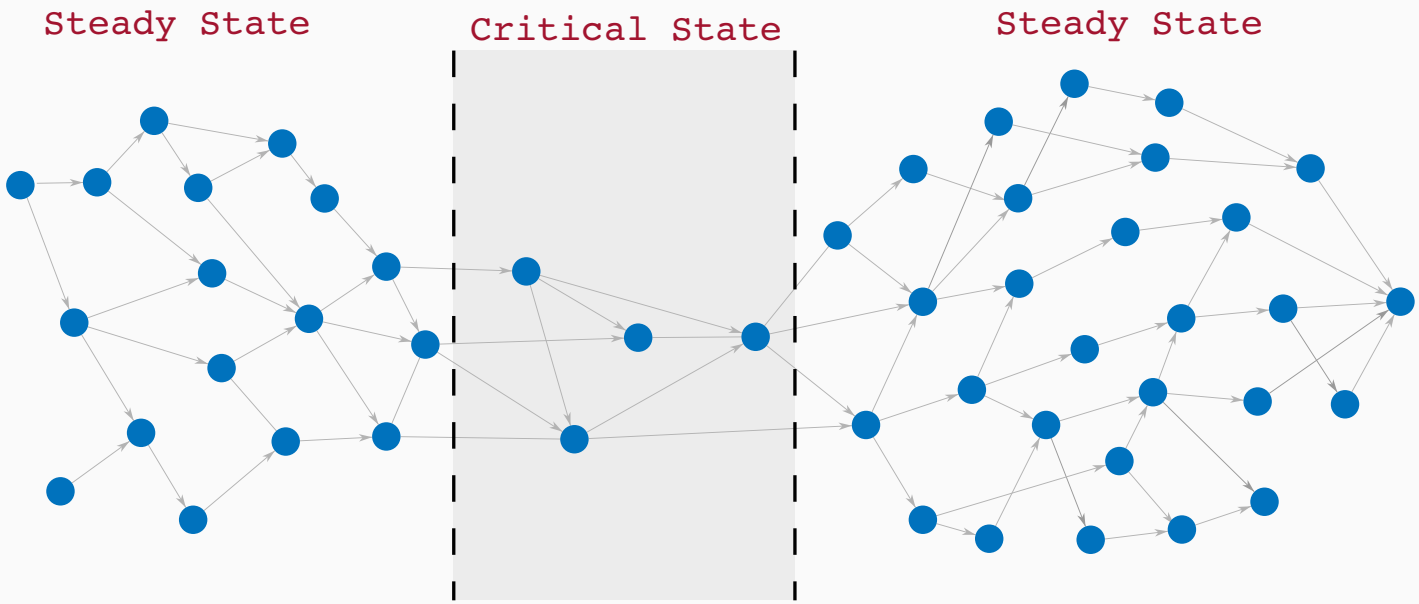


Building Blocks

- A task container storing tasks ready for execution.
- Two methods:
  - **push**: put ready tasks in the scheduler.
  - **pop** : retrieve ready tasks from scheduler.

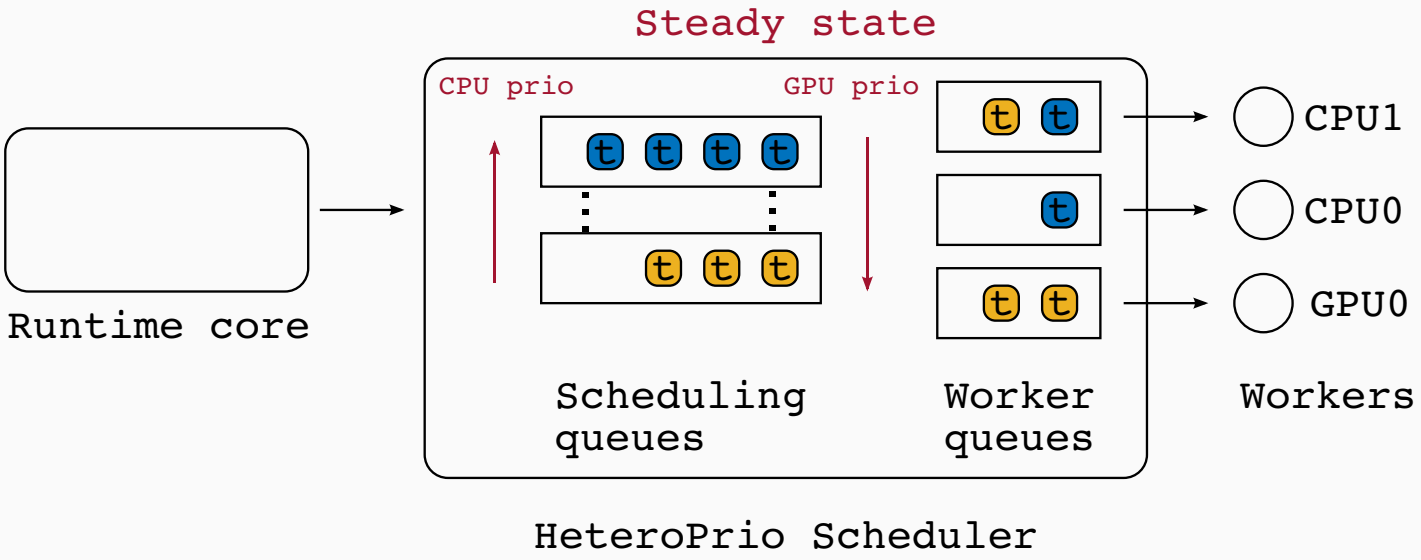


DAGs may be irregular and alternate rich and poor concurrency regions.



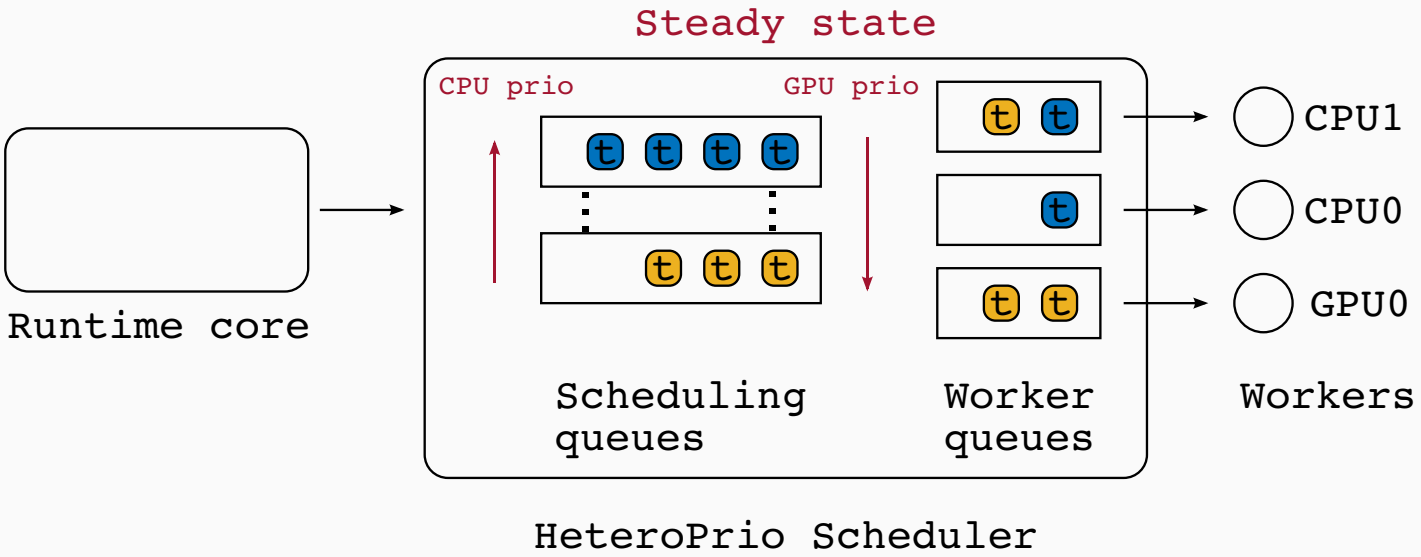
DAGs may be irregular and alternate rich and poor concurrency regions. Our scheduler switches automatically between two states:

- Steady-state: # of ready tasks >> number of resources: execute tasks where they are best suited i.e. best acceleration factor.
- Critical-state: # of ready tasks << number of resources: reduce the time spent on the critical path.



HeteroPrio [1, 4]: Steady state

- Tasks are prioritized on each type of resources according to their acceleration factor.
- Tasks are put in scheduling queues at push operation and moved to worker queues at pop.

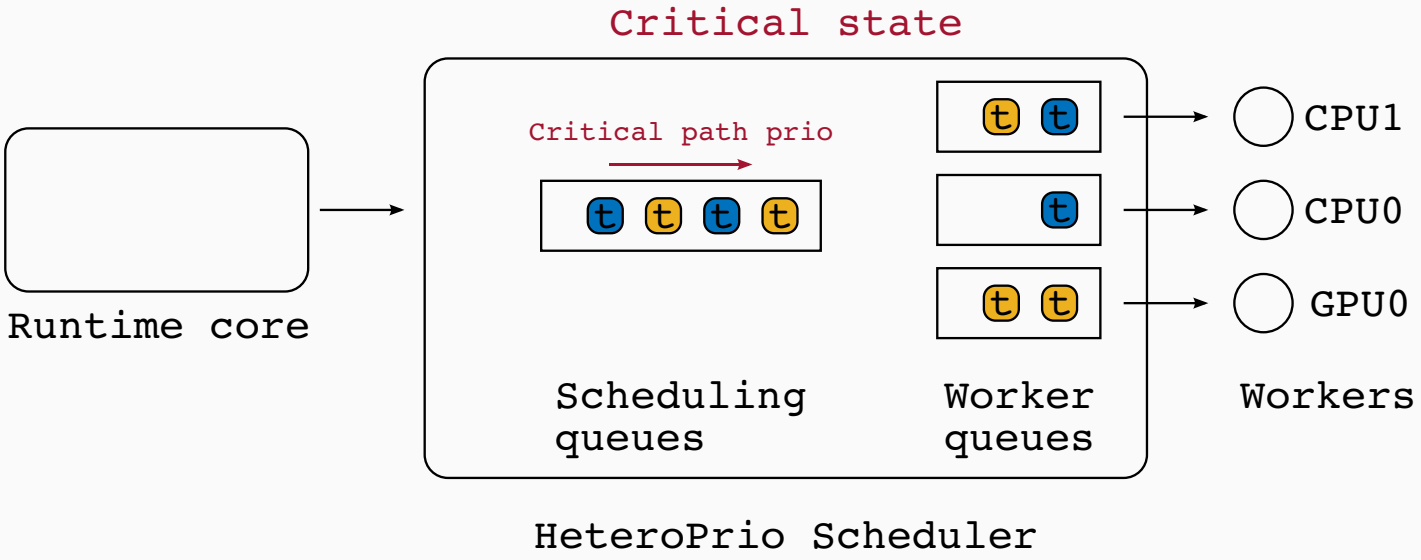


HeteroPrio [1, 4]: Steady state

- Tasks are prioritized on each type of resources according to their acceleration factor.
- Tasks are put in scheduling queues at push operation and moved to worker queues at pop.

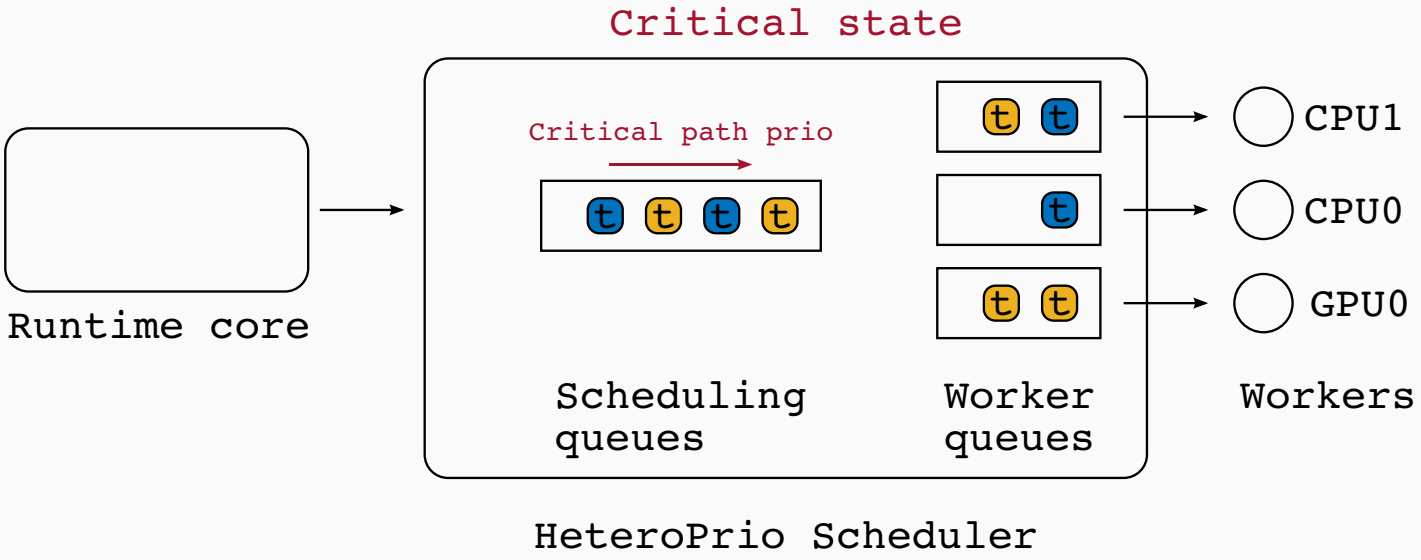
▲ Good matching between tasks and units.

▲ Data prefetching thanks to worker queues.



HeteroPrio [1, 4]: Critical state

- Tasks are prioritized regarding the critical path.



HeteroPrio [1, 4]: Critical state

- Tasks are prioritized regarding the critical path.

▲ Avoids stalls in the pipeline.

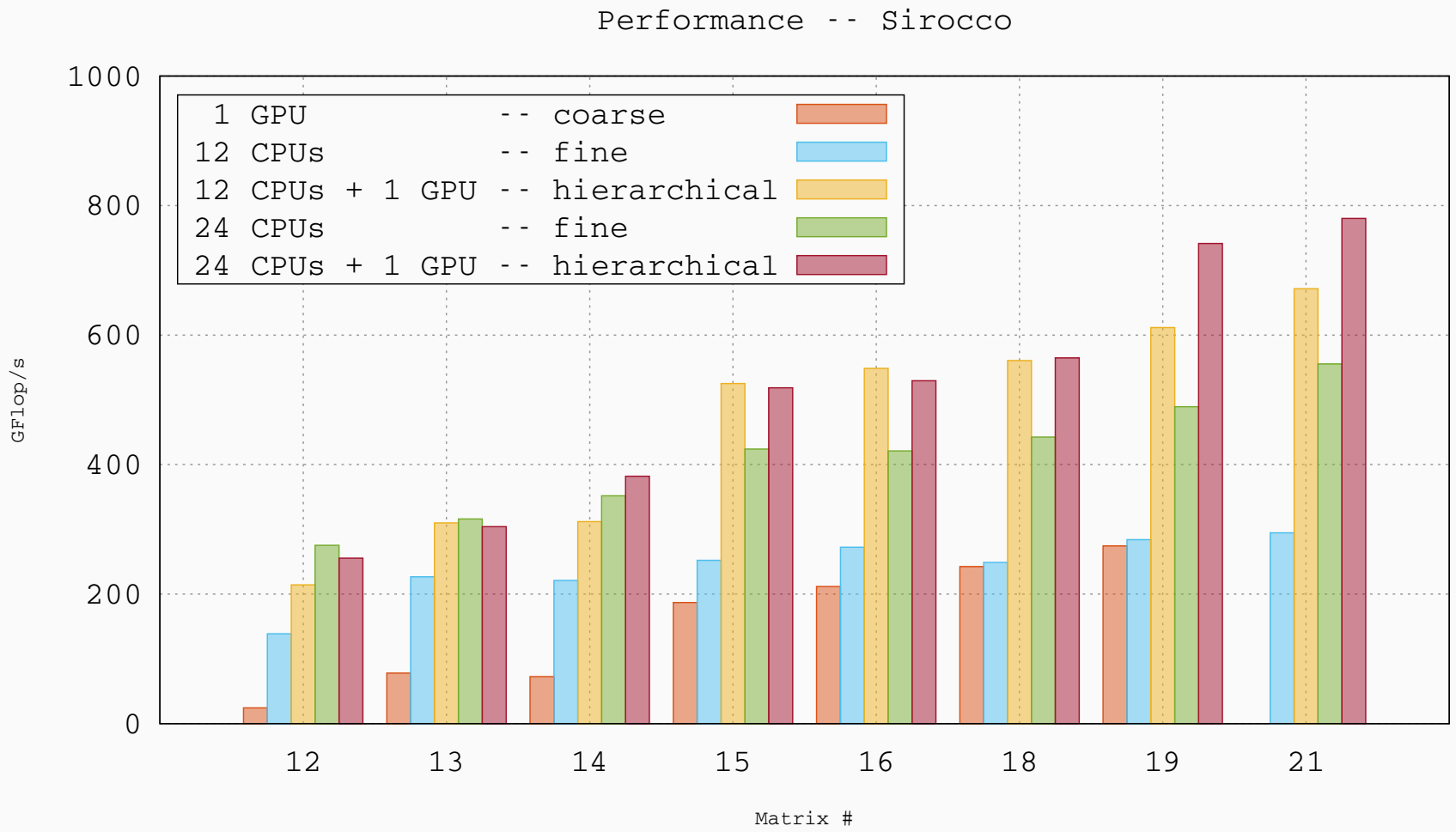
▲ Data prefetching.

#	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

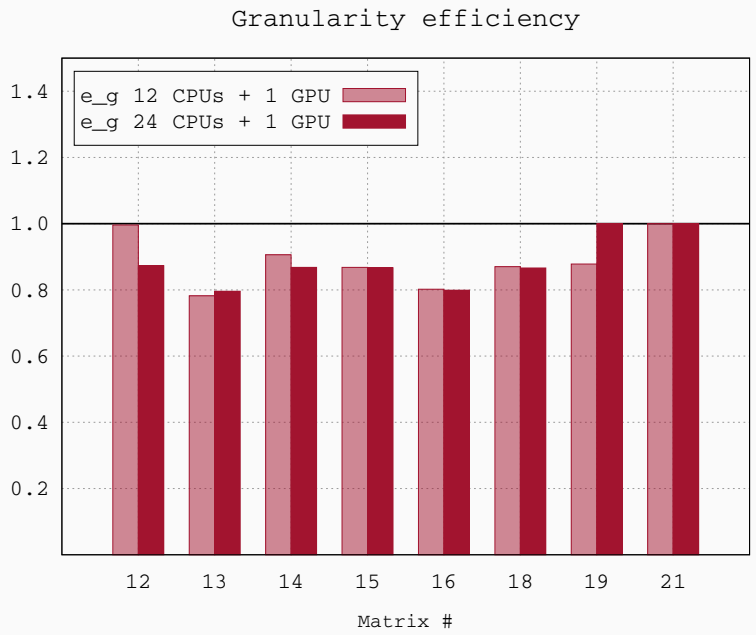
System Sirocco:

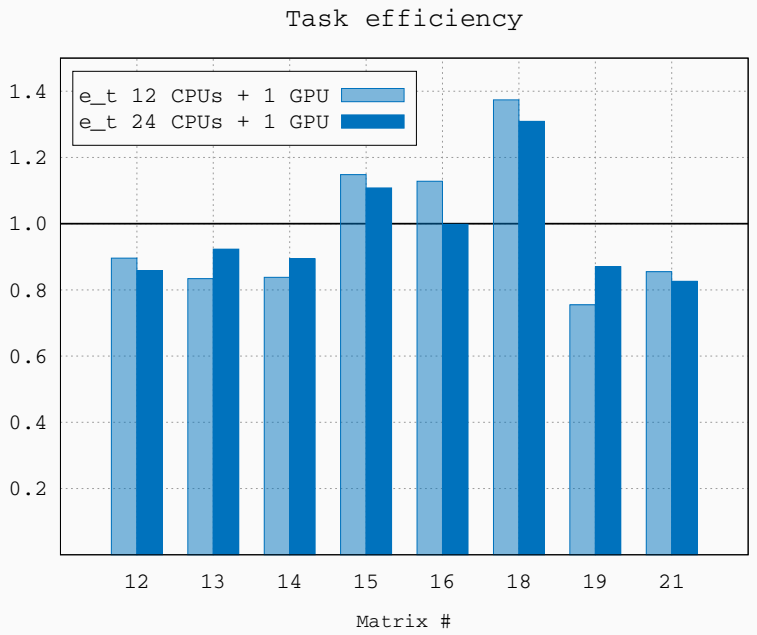
- Haswell Intel Xeon E5-2680 @ 2.5 GHz, 2×12 cores
- 128 GB memory (NUMA)

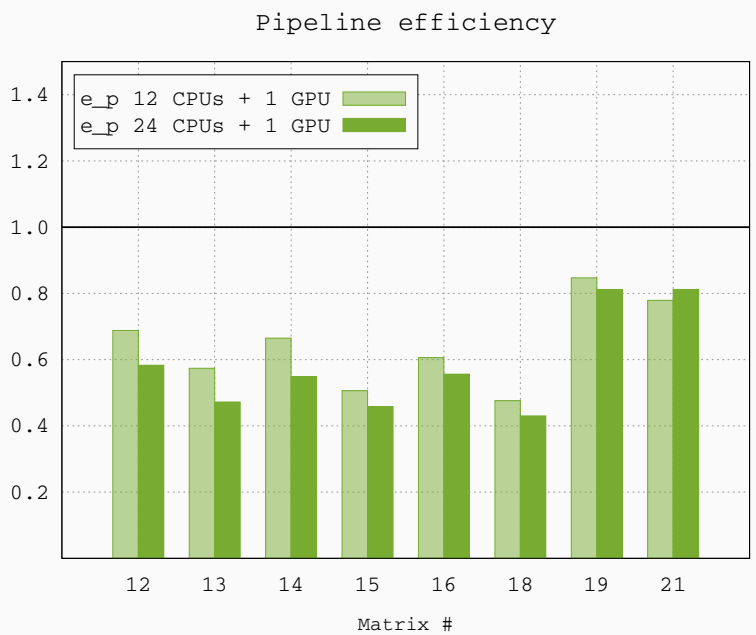
EXPERIMENTAL RESULTS: ABSOLUTE PERFORMANCE

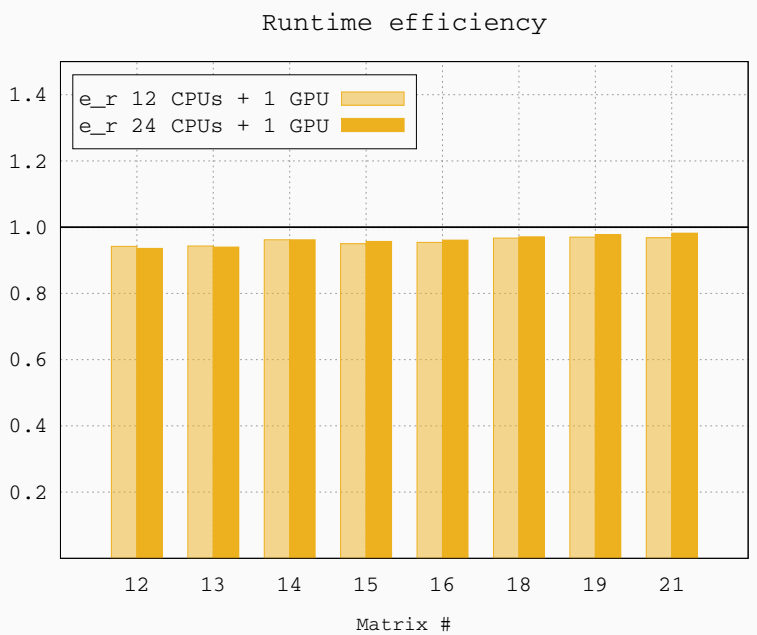


EXPERIMENTAL RESULTS: EFFICIENCY BREAKDOWN

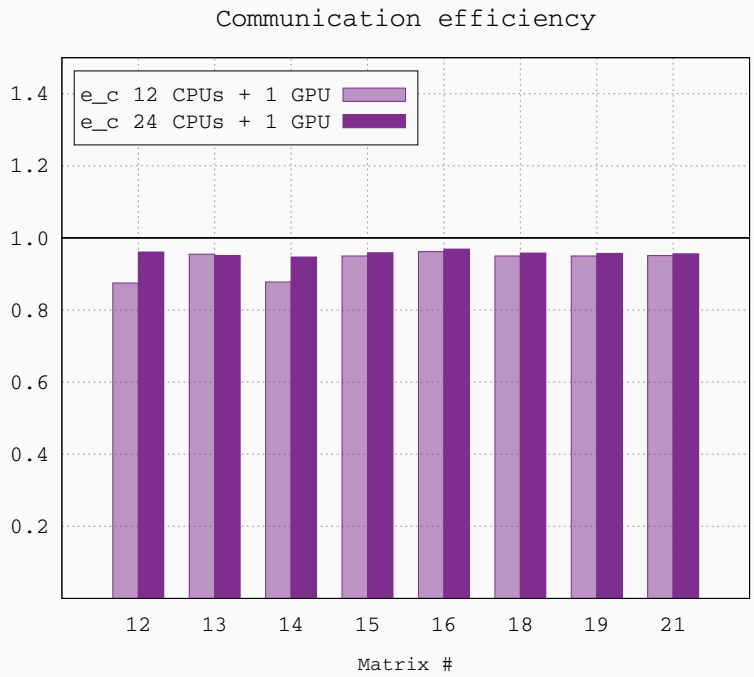


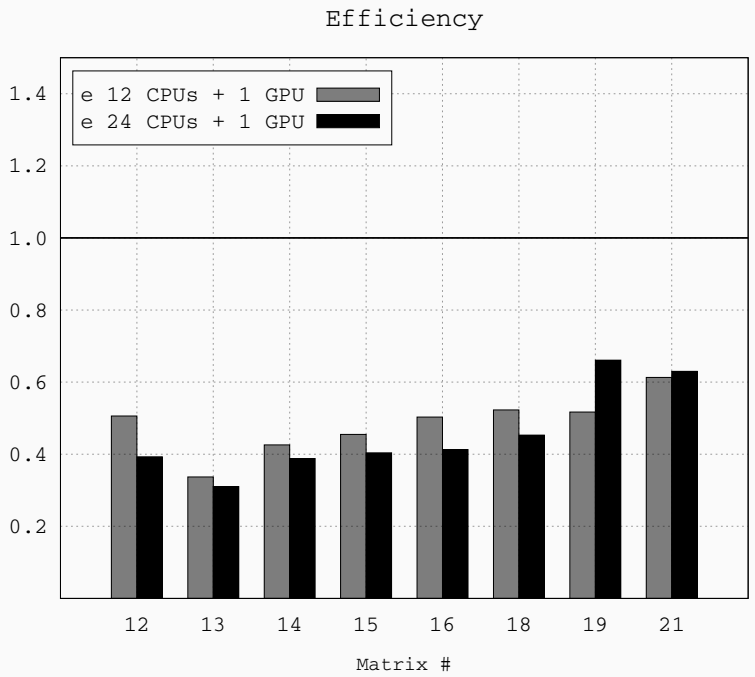




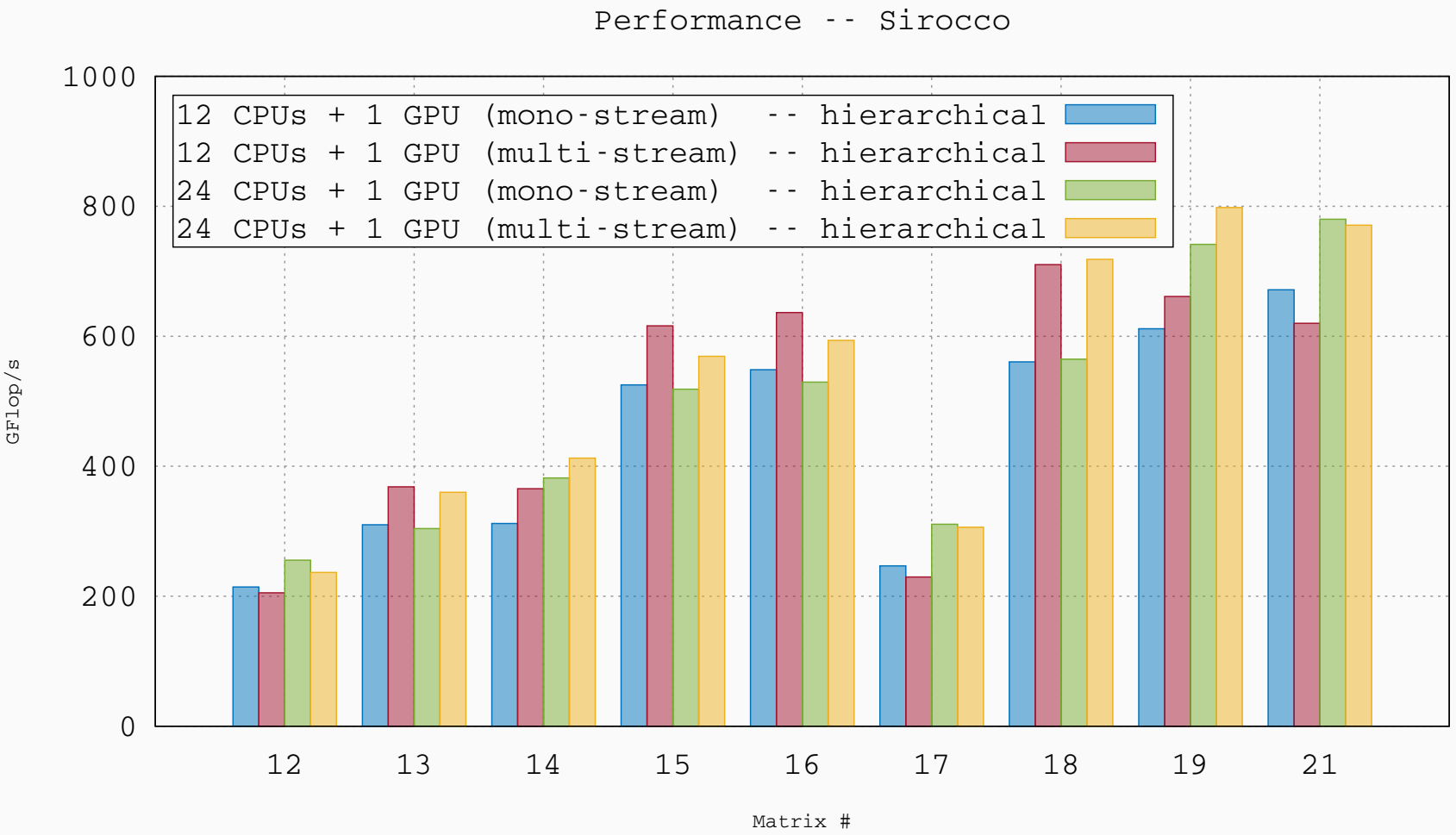


EXPERIMENTAL RESULTS: EFFICIENCY BREAKDOWN



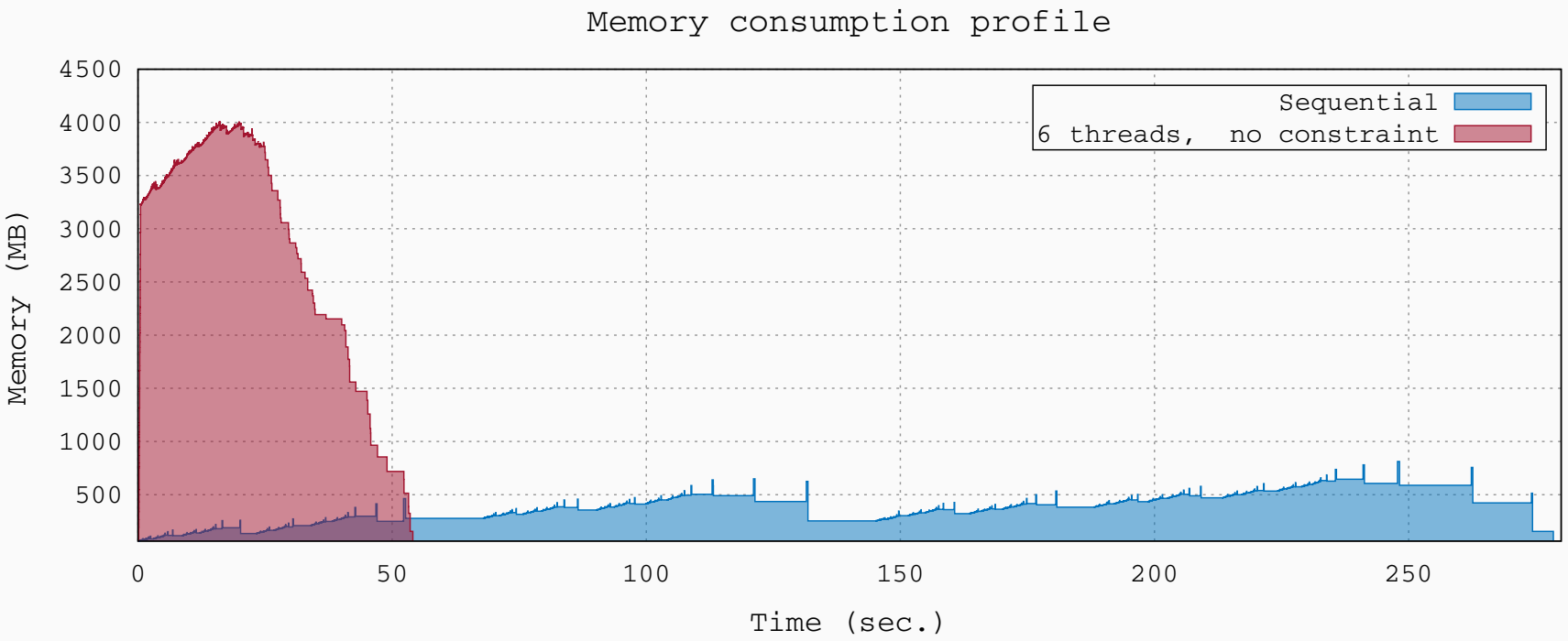


EXPERIMENTAL RESULTS: ABSOLUTE PERFORMANCE



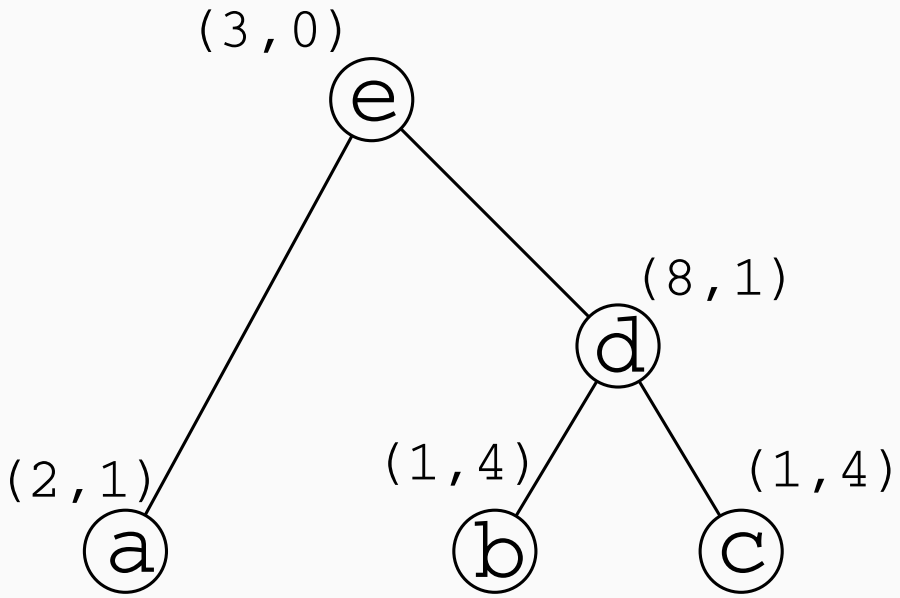
MEMORY-AWARE MULTIFRONTAL METHOD

MEMORY FOOTPRINT IN THE MULTIFRONTAL METHOD



- Tree-level parallelism increases the memory consumption.
- During a parallel execution, the memory consumption is generally much bigger than the sequential memory consumption.

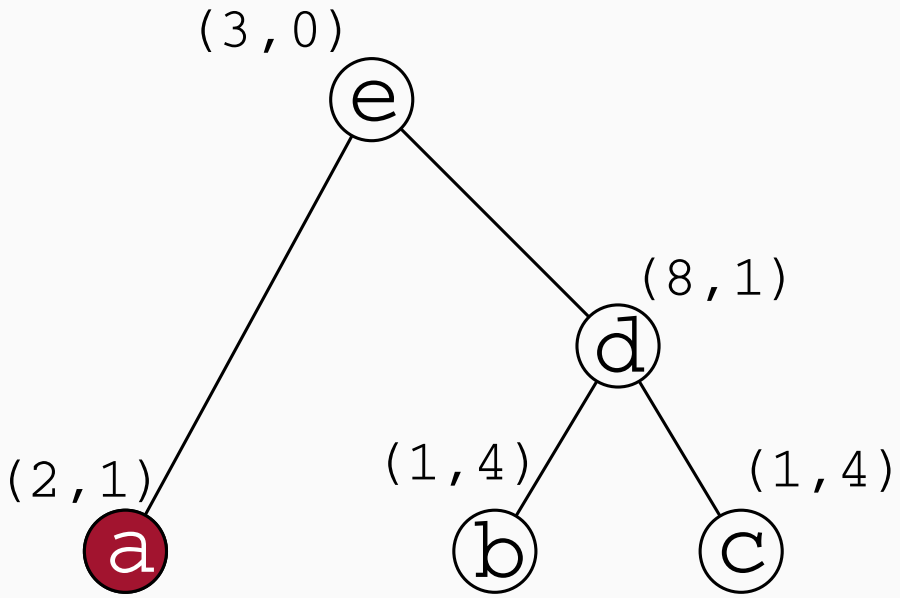
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory

Memory peak during the sequential tree traversal:

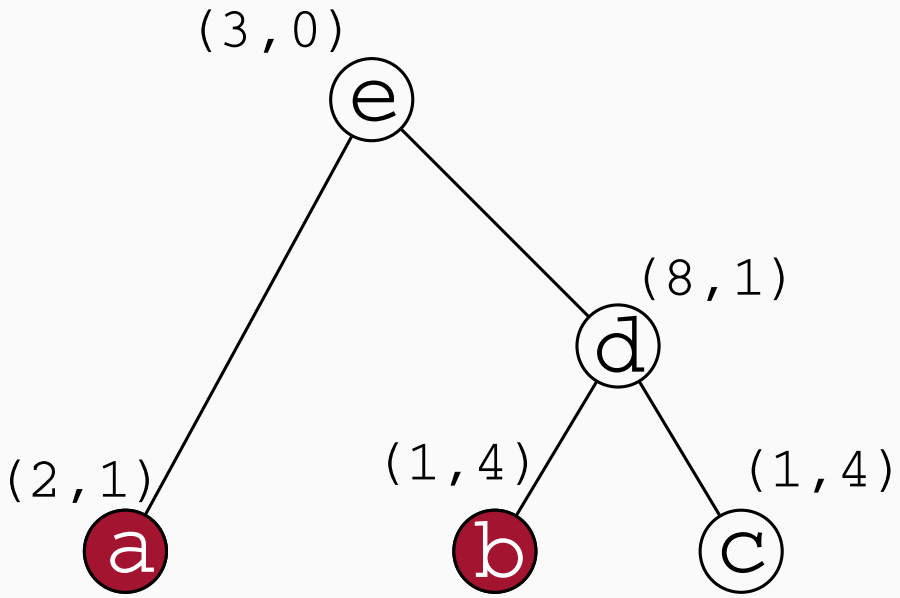
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3

Memory peak during the sequential tree traversal:

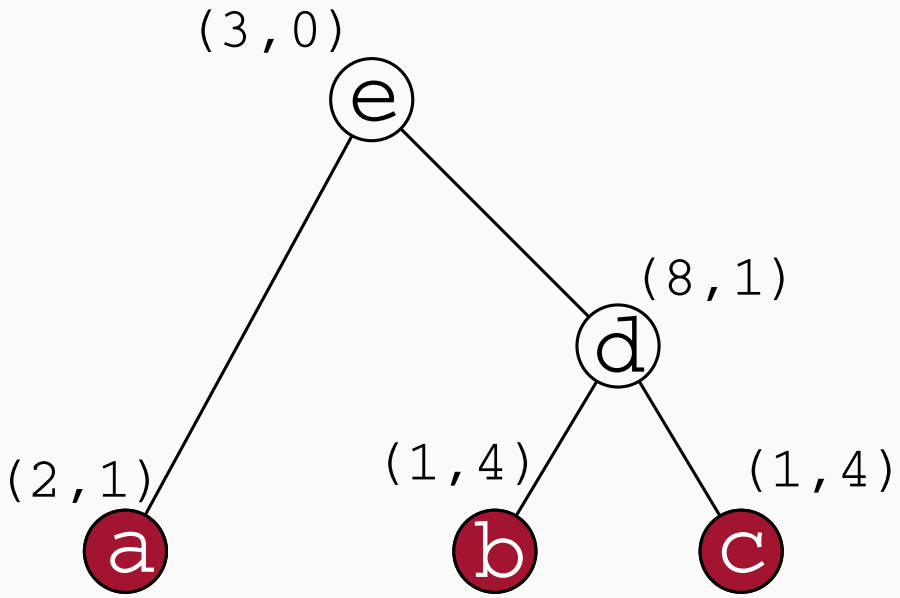
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3
`activate(b)`	8

Memory peak during the sequential tree traversal:

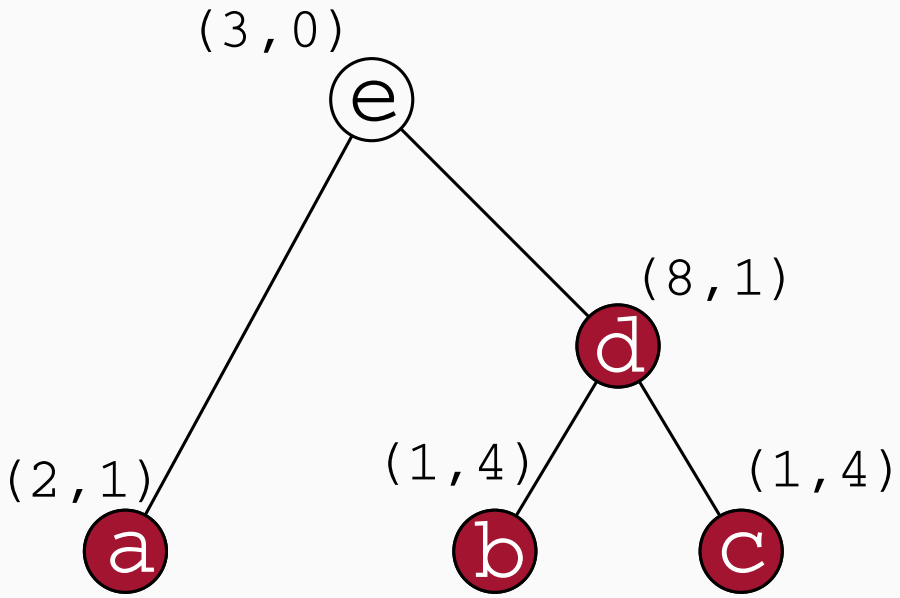
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3
`activate(b)`	8
`activate(c)`	13

Memory peak during the sequential tree traversal:

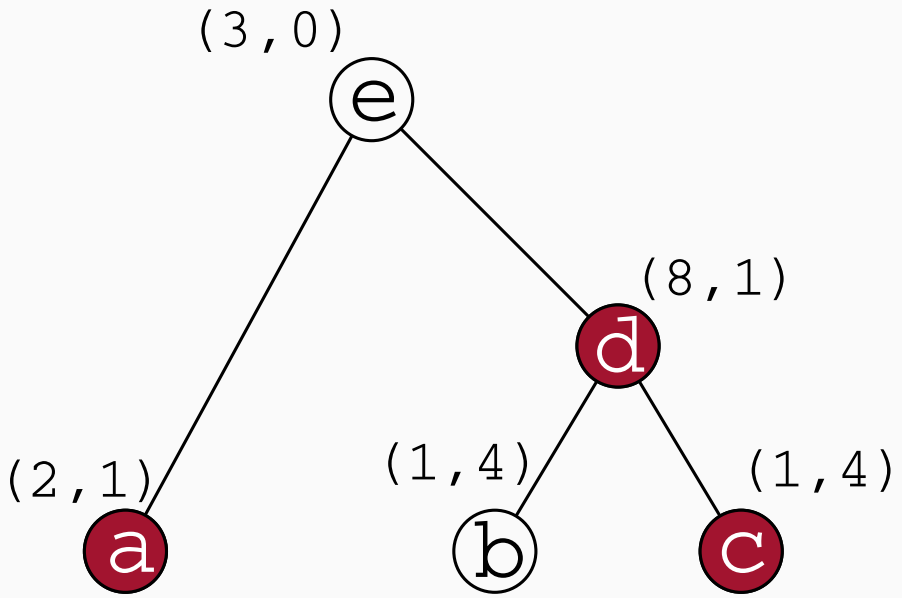
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3
`activate(b)`	8
`activate(c)`	13
`activate(d)`	22

Memory peak during the sequential tree traversal:

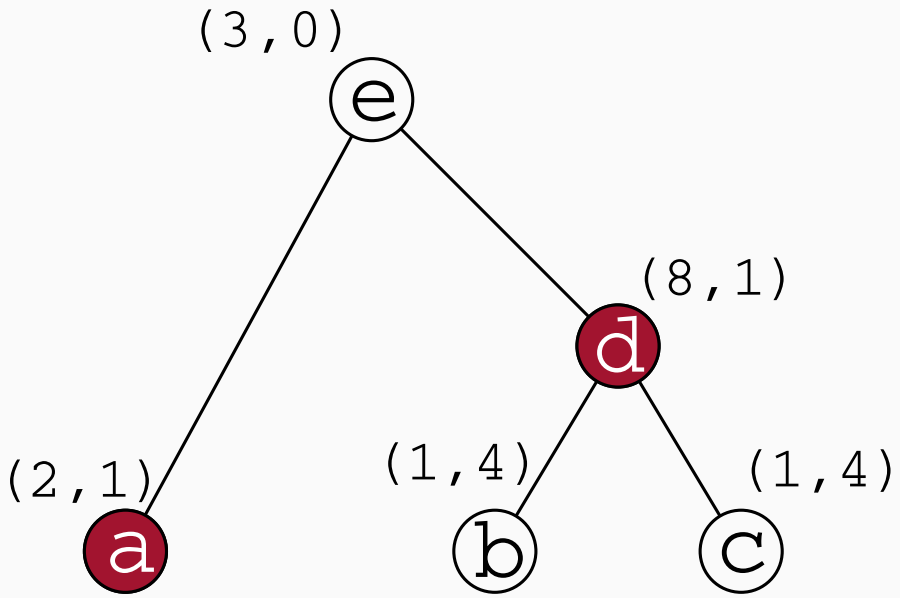
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3
`activate(b)`	8
`activate(c)`	13
`activate(d)`	22
`deactivate(b)`	18

Memory peak during the sequential tree traversal:

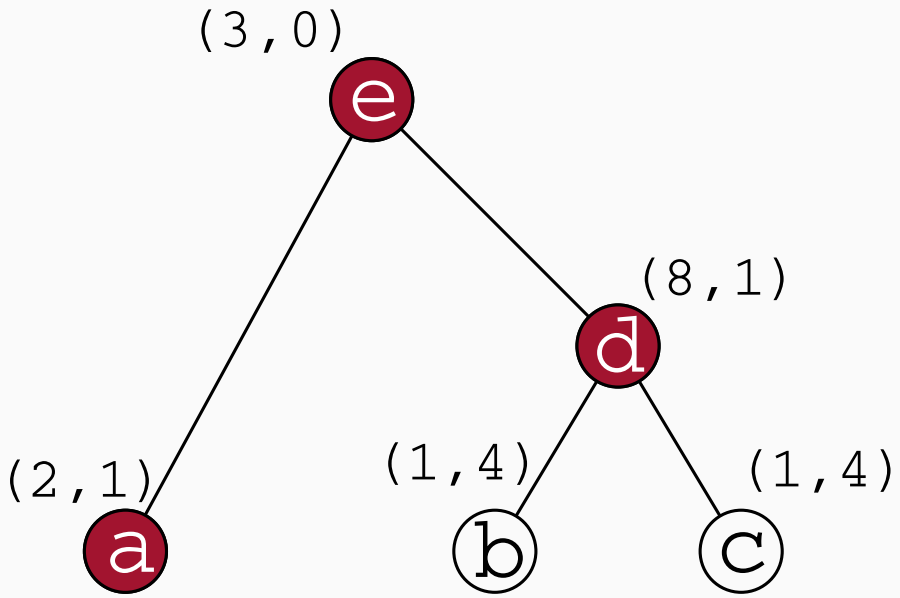
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
`activate(a)`	3
`activate(b)`	8
`activate(c)`	13
`activate(d)`	22
`deactivate(b)`	18
`deactivate(c)`	14

Memory peak during the sequential tree traversal:

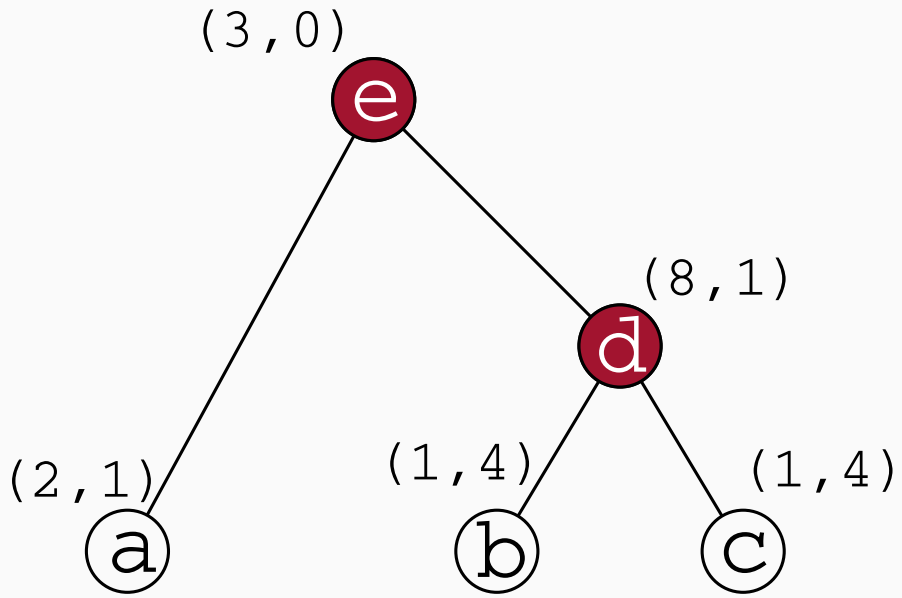
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	22
deactivate(b)	18
deactivate(c)	14
activate(e)	17

Memory peak during the sequential tree traversal:

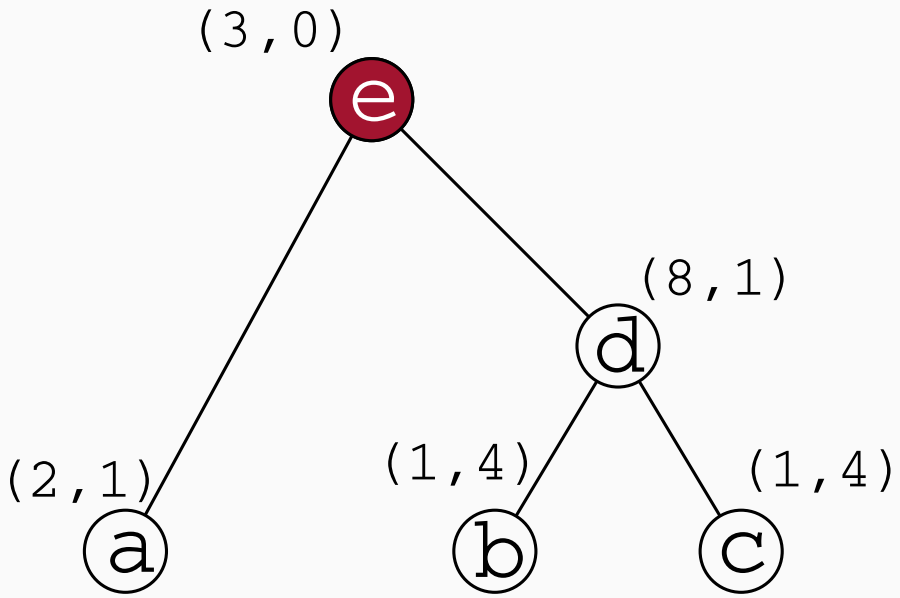
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	22
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16

Memory peak during the sequential tree traversal:

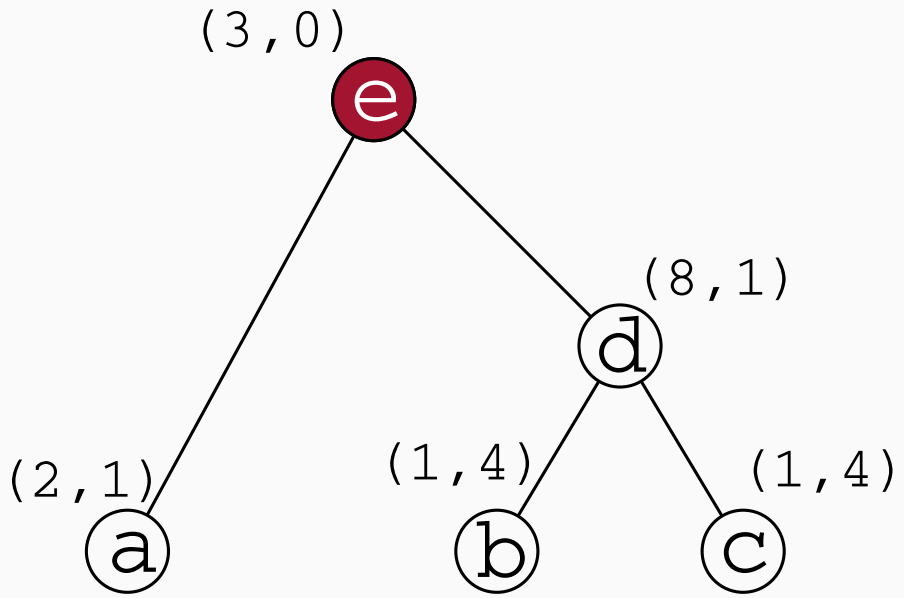
IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	22
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16
deactivate(d)	15

Memory peak during the sequential tree traversal:

IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION

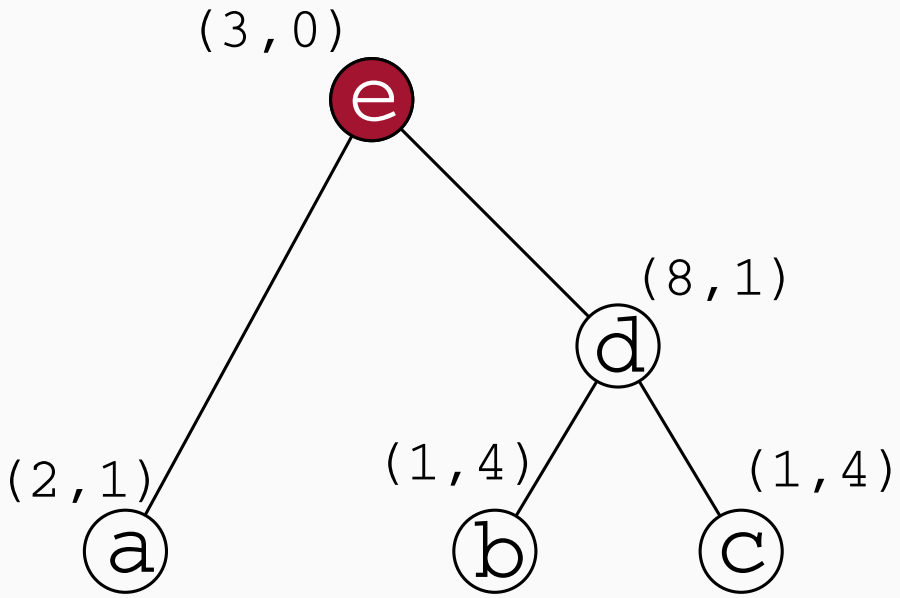


Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	22
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16
deactivate(d)	15

Memory peak during the sequential tree traversal:

- natural tree traversal order a-b-c-d-e: 22 memoy units

IMPACT OF THE TREE TRAVERSAL ON THE MEMORY CONSUMPTION



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	22
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16
deactivate(d)	15

Memory peak during the sequential tree traversal:

- natural tree traversal order a-b-c-d-e: 22 memoy units
- optimal tree traversal order b-c-d-a-e: 19 memoy units

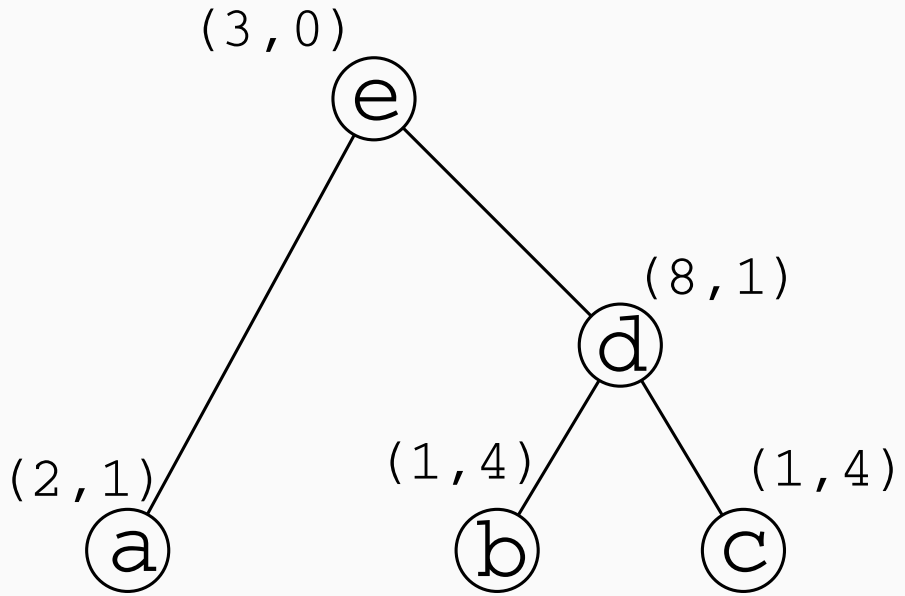
Problem: minimize the memory footprint of the factorization.
Sequential case:

- Memory-minimizing postorder traversal: Liu's algorithms [7]

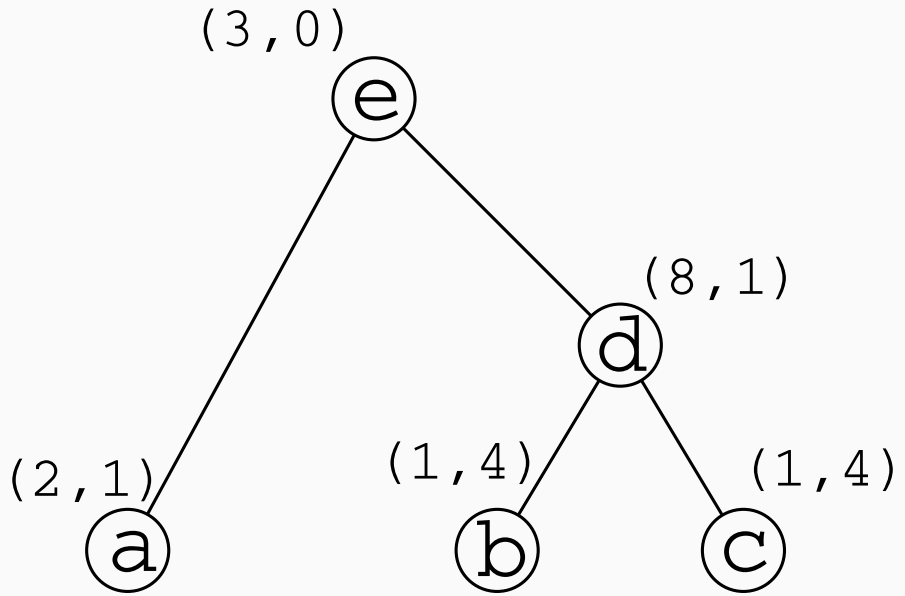
Parallel case:

- The problem is NP-complete and no approximation algorithms can be designed to tackle the problem [6].
- Eyraud-Dubois et al. [6] propose several heuristics such as MemBookingInnerFirst for the scheduling of task trees under a given memory constraint.

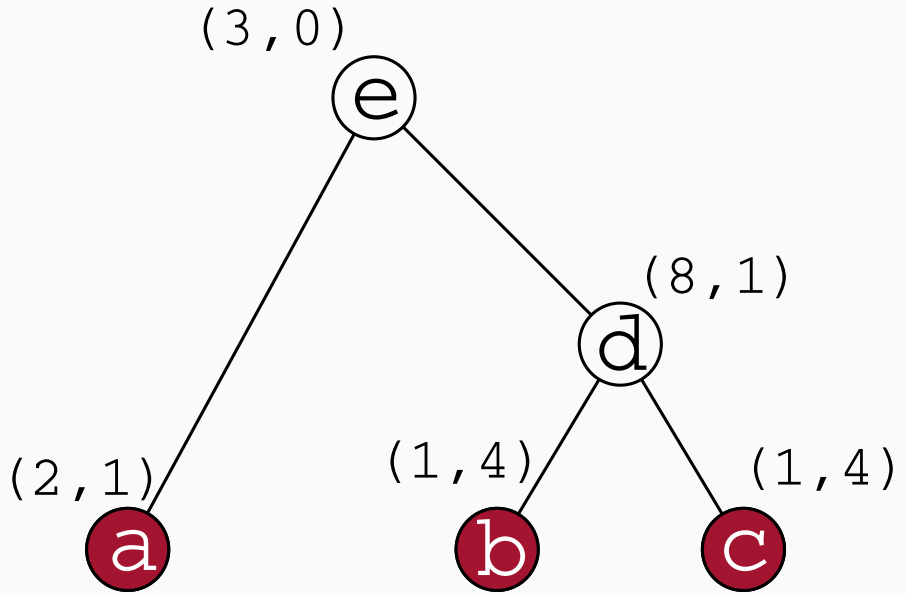
Objective: limit the memory footprint of the parallel factorization.



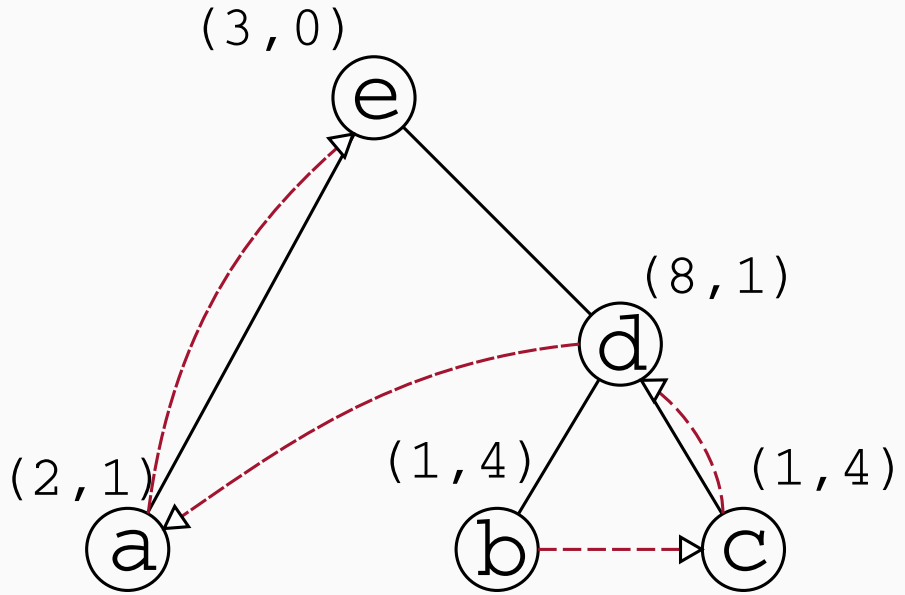
Objective: limit the memory footprint of the parallel factorization.
Example: Memory constraint: 19 memory units.



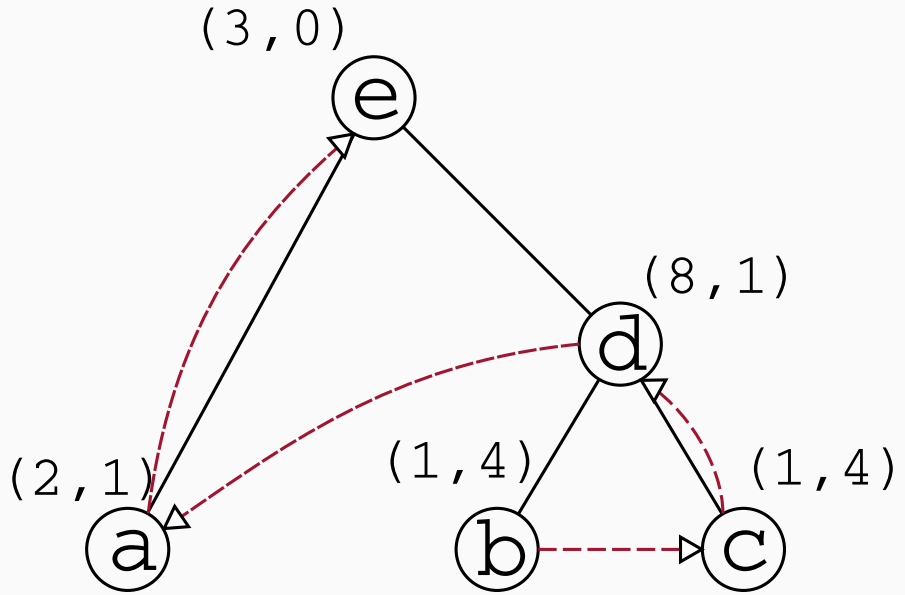
Objective: limit the memory footprint of the parallel factorization.
Example: Memory constraint: 19 memory units. Let's assume that nodes a, b and c are activated in parallel consuming 13 meory units and node d cannot be allocated: Deadlock.



Objective: limit the memory footprint of the parallel factorization.
Approach: the activation of fronts is forced to respect the memory-minimizing sequential order. The tasks submitted as a results of a front activation can be executed in any order.



Objective: limit the memory footprint of the parallel factorization.
Approach: the activation of fronts is forced to respect the memory-minimizing sequential order. The tasks submitted as a results of a front activation can be executed in any order.



This ensures that the parallel execution runs within the same memory envelop as the sequential one. The memory constraint can be relaxed in order to permit the activation of more fronts and, potentially achieve more concurrency.

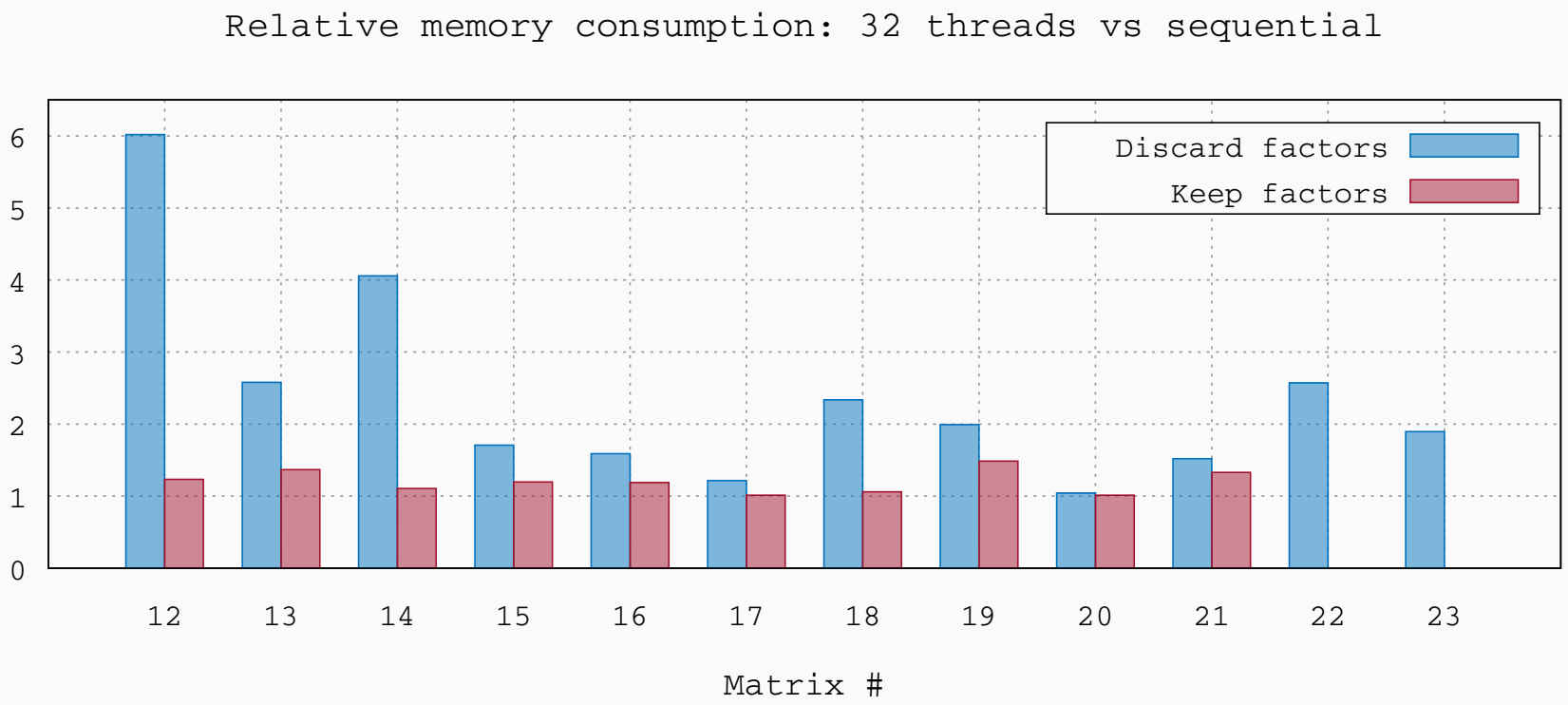
MEMORY AWARE SCHEDULING IN A STF MODEL

```
do f=1, nfronts ! in postorder

   call activate(f)    ! compute structure and register handles

   call submit(init, f) ! allocate and initialize front

   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, f)
      end do
      ! cleanup child
      call submit(clean, c)
   end do

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

MEMORY AWARE SCHEDULING IN A STF MODEL

```
do f=1, nfronts ! in postorder
   do while(avail_mem < size(f)) wait()

   call activate(f) ! compute structure and register handles

   call allocate(f)    ! allocate front: avail_mem -= size (f)

   call submit(init, f) ! initialize front

   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, f)
      end do
      ! cleanup child: avail_mem += size ( cb ( f ) )
      call submit(clean, c)
   end do

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

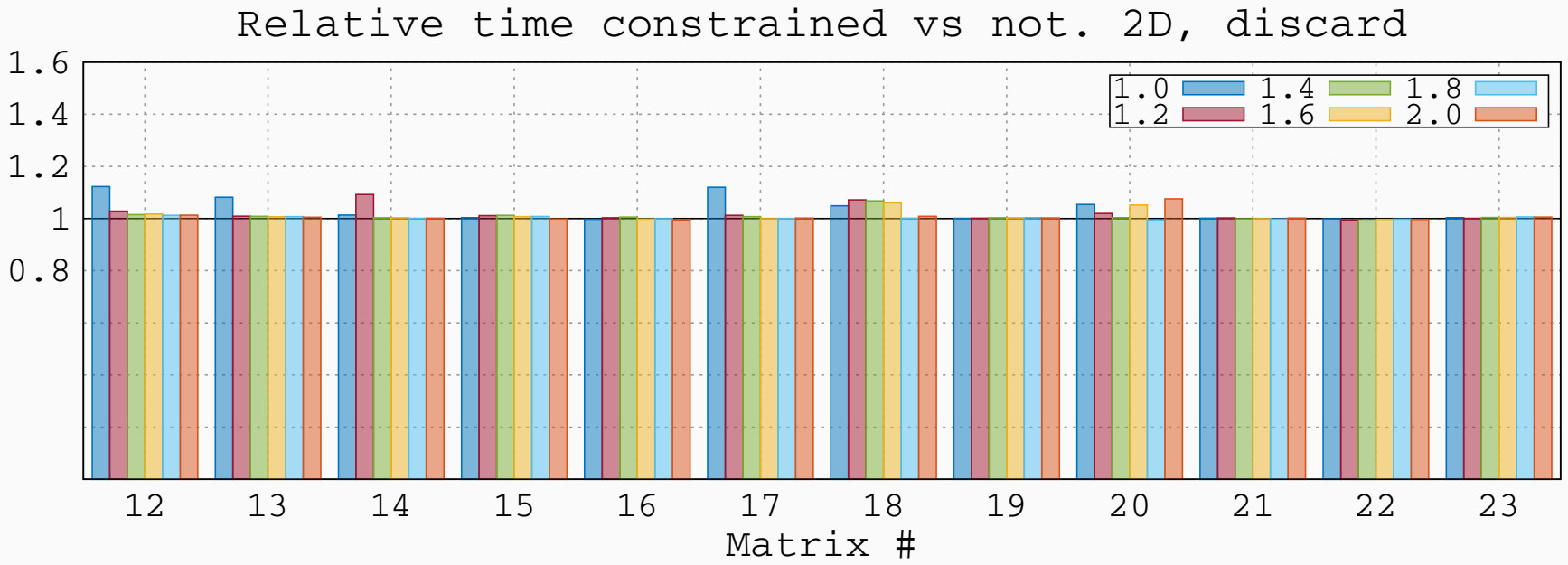
EXPERIMENTAL RESULTS: MEMORY CONSTRAINED FACTORIZATION

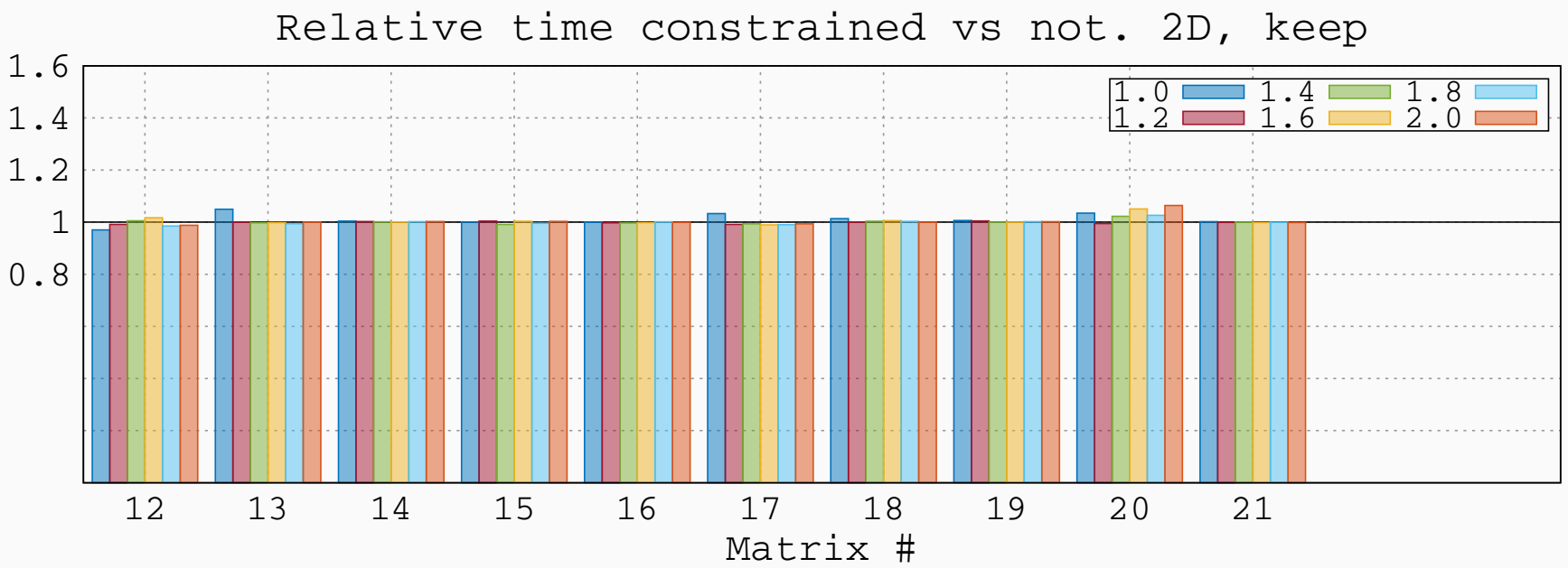


Two scenarios for the parallel execution:

- In-Core (IC) execution: this is the most common case where the computed factors are kept in memory.
- Out-Of-Core (OOC) execution: in this scenario the factors are written to disk as they are computed in order to save memory.

EXPERIMENTAL RESULTS: MEMORY CONSTRAINED FACTORIZATION





CONCLUSION AND FUTURE WORK

Conclusions

- STF is suited to complex workloads and algorithms such as sparse matrix factorizations.
- The performance of the parallel execution for the STF-based multifrontal factorization under a memory constraint is as high as the unconstrained case.
- STF eases the implementation of GPU-based multifrontal method.
- Modern runtime engines are efficient and introduce a small (almost insignificant) overhead.

On-going and future work

- Multi-GPU architectures.
- Distributed memory architectures.

Thanks!

Questions?

[1] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. *Task-based FMM for heterogeneous architectures*. Research Report RR-8513. Inria, Apr. 2014, p. 29. URL: https://hal.inria.fr/hal-00974674.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*. Tech. rep. IRI/RT–2014-03–FR. Submitted to ACM Transactions On Mathematical Software. IRIT, Nov. 2014. URL: http://buttari.perso.enseeiht.fr/stuff/IRI-RT--2014-03--FR.pdf.

[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*. Tech. rep. IRI/RT–2015–02–FR. Accepted at the HiPC 2015 conference. IRIT, June 2015. URL: https://hal.archives-ouvertes.fr/hal-01166312v2.

[5] A. Buttari. "Fine granularity sparse QR factorization for multicore based systems". In: *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*. PARA'10. Reykjavik, Iceland: Springer-Verlag, 2012, pp. 226–236. ISBN: 978-3-642-28144-0. URL: http://dx.doi.org/10.1007/978-3-642-28145-7_23.

[6] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. "Parallel Scheduling of Task Trees with Limited Memory". In: *ACM Trans. Parallel Comput. 2.2* (June 2015), 13:1–13:37. ISSN: 2329-4949. DOI: 10.1145/2779052. URL: http://doi.acm.org/10.1145/2779052.

[7] J. W. H. Liu. "On the storage requirement in the out-of-core multifrontal method for sparse factorization". In: *ACM Transactions On Mathematical Software* 12 (1986), pp. 127–148.