Scheduling recursive tasks on homogeneous and heterogeneous systems

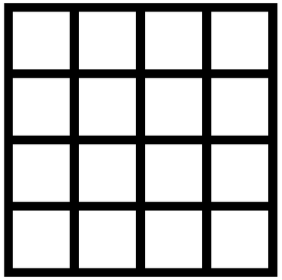
Thomas Morin, Nathalie Furmento, Abdou Guermouche,
Samuel Thibault, Pierre-André Wacrenier

INRIA Bordeaux - Sud-Ouest -- STORM Team

Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

🟦 POTRF

🟪 TRSM

■ SYRK

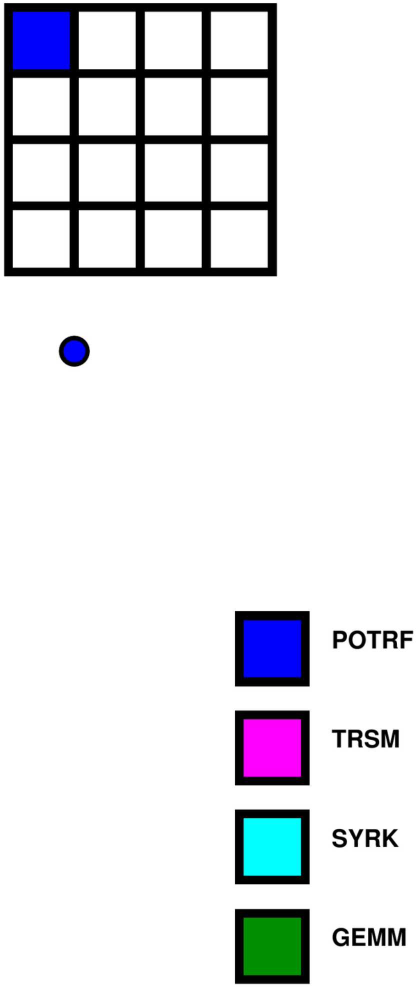
🟩 GEMM

Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

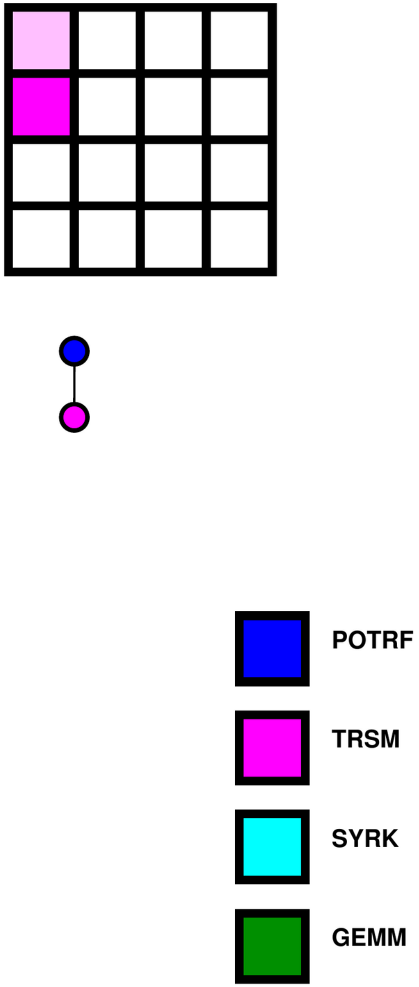


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

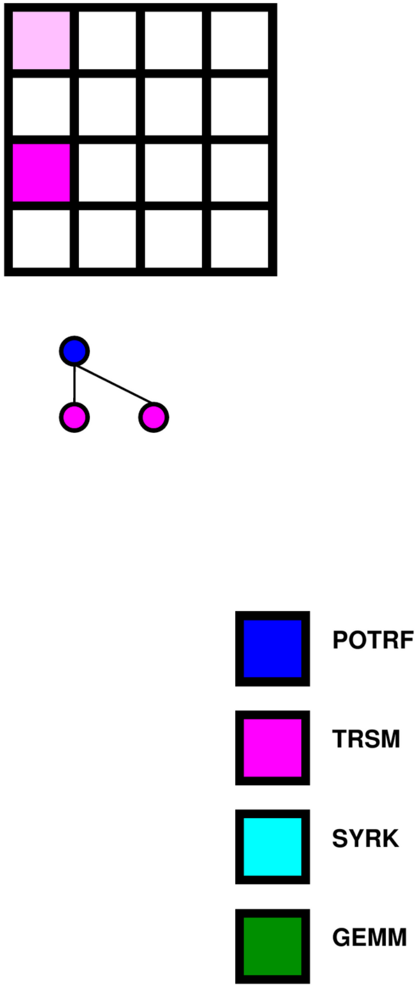
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

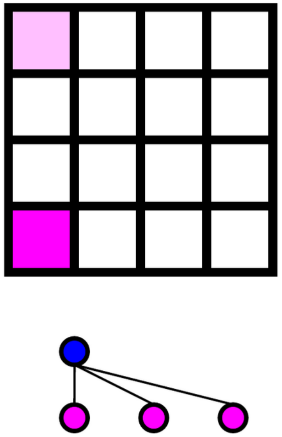


```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

🟦 POTRF

🟪 TRSM

🟦 SYRK

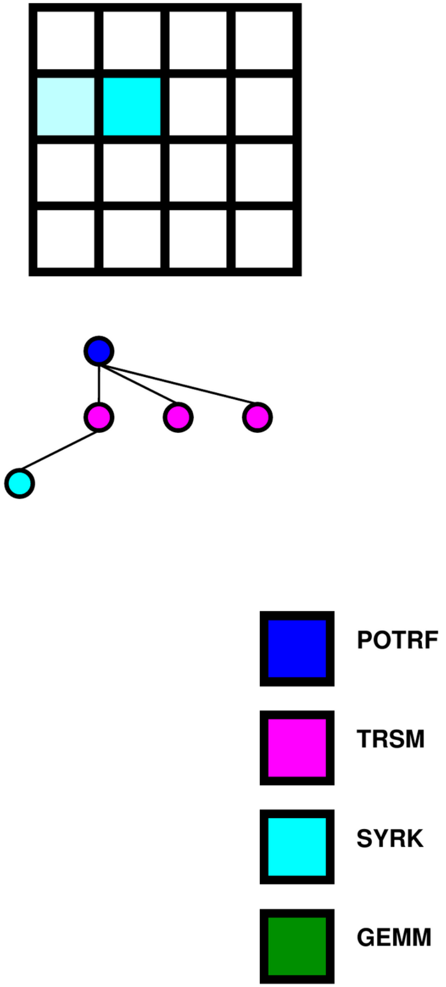
🟩 GEMM

Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

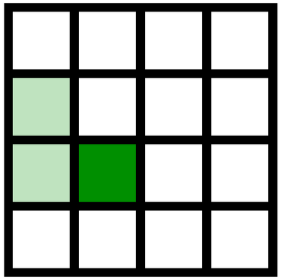
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



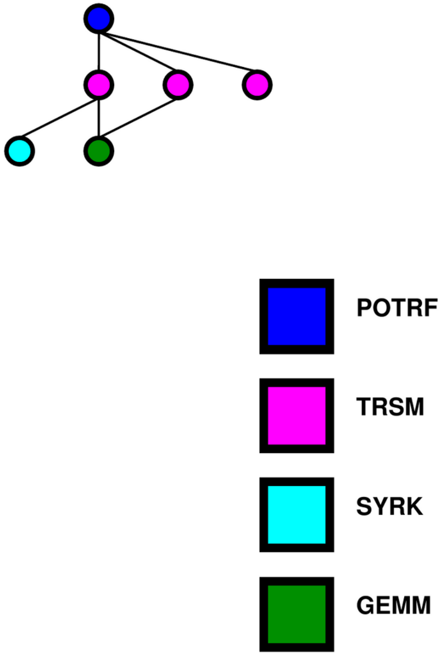
Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



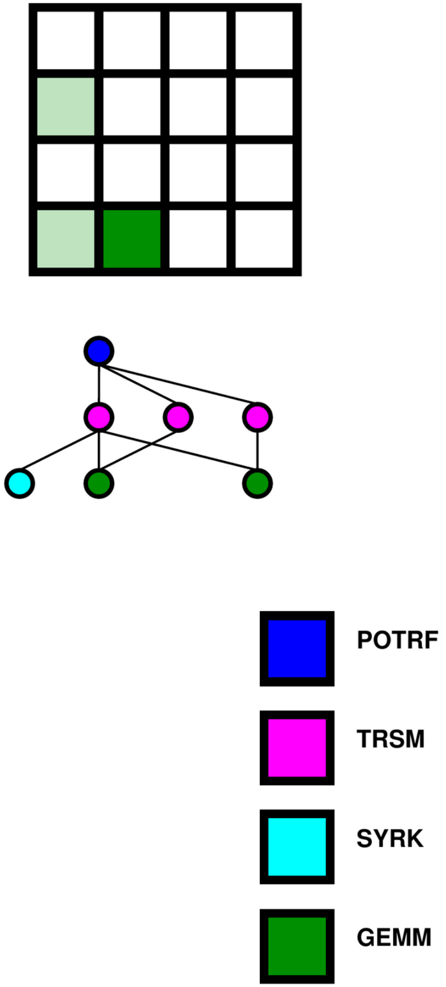
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



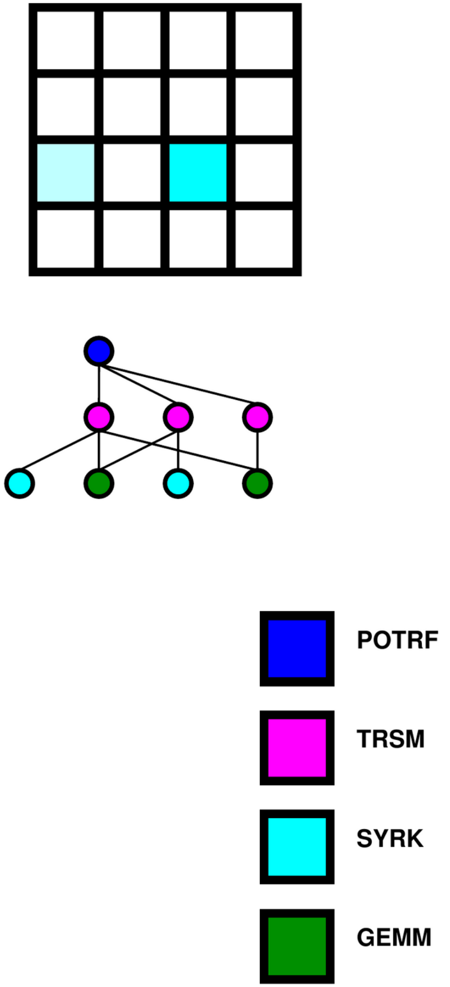
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

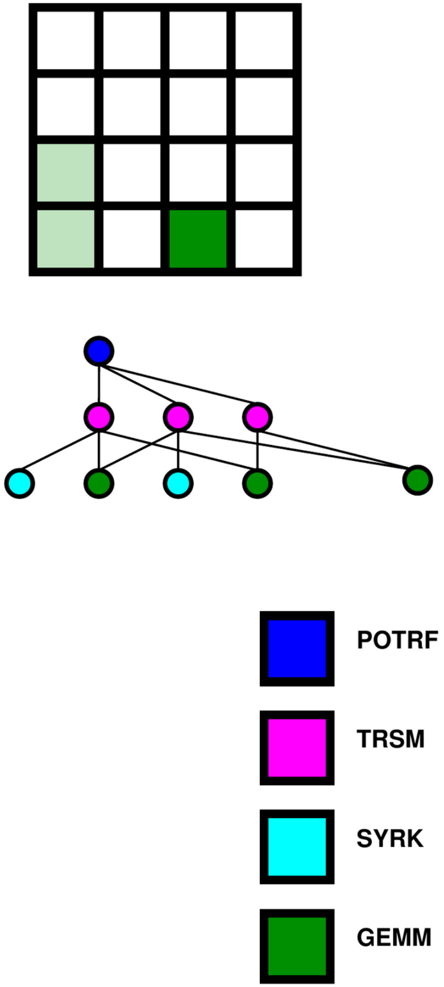


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

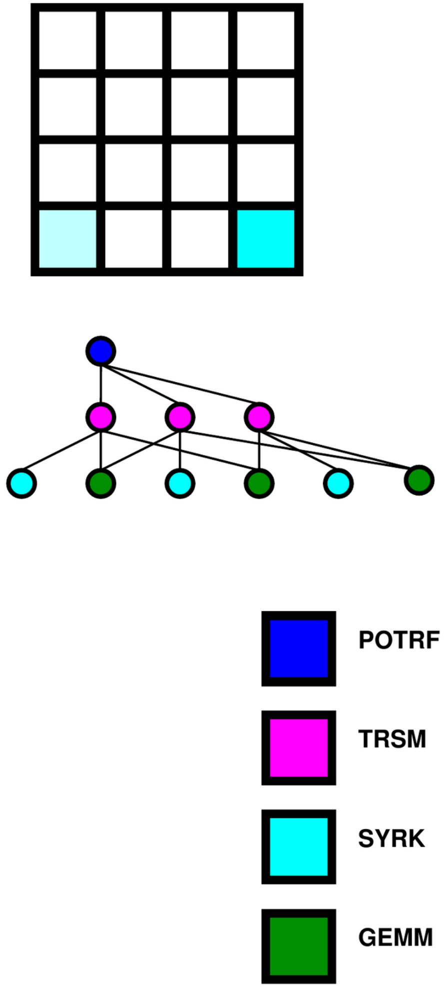


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

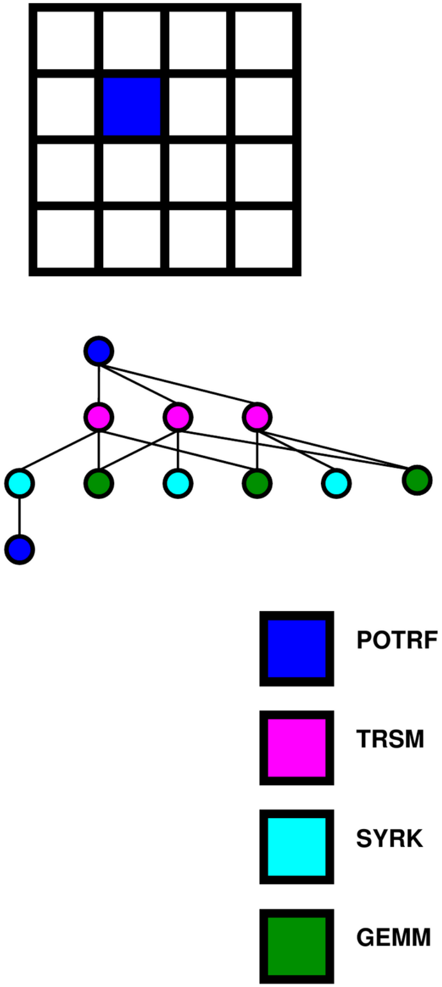


Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

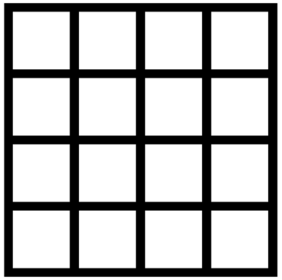
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



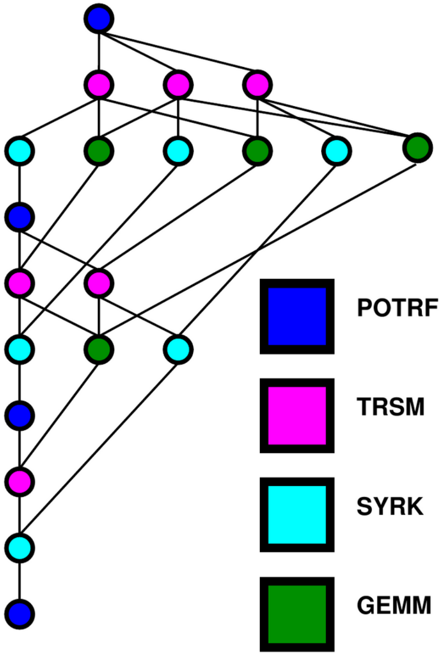
Expressing a task graph

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)



```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

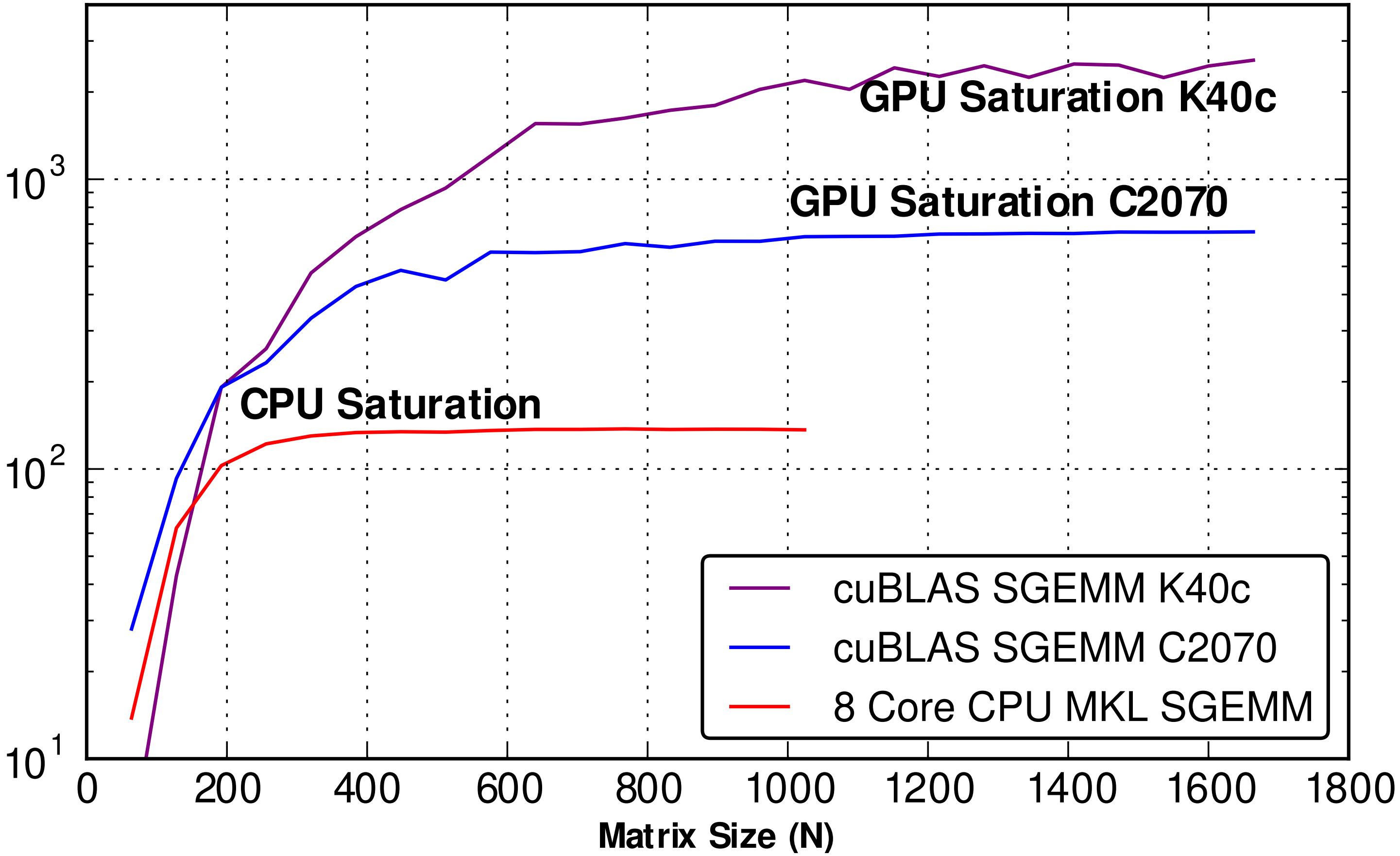


How big should a task be?

How big should a task be?

- Small enough to get parallelism to **feed** all processing units
- Large enough to **efficiently** use the processing units

How big should a task be?



From PARSEC : « Hierarchical DAG Scheduling for Hybrid Distributed Systems », Wu, Bouteiller, Bosilca, Faverge, Dongarra

How big should a task be?

GPUs

- Have **thousands** of cores to feed
- Newer generations require yet larger sizes
- Can run several kernels at the same time
  - Still limited

How big should a task be?

CPUs

- Have many independent cores

  → Need many tasks

- Can use parallel implementations (e.g. from MKL)
  - But better have subtasks to interleave them

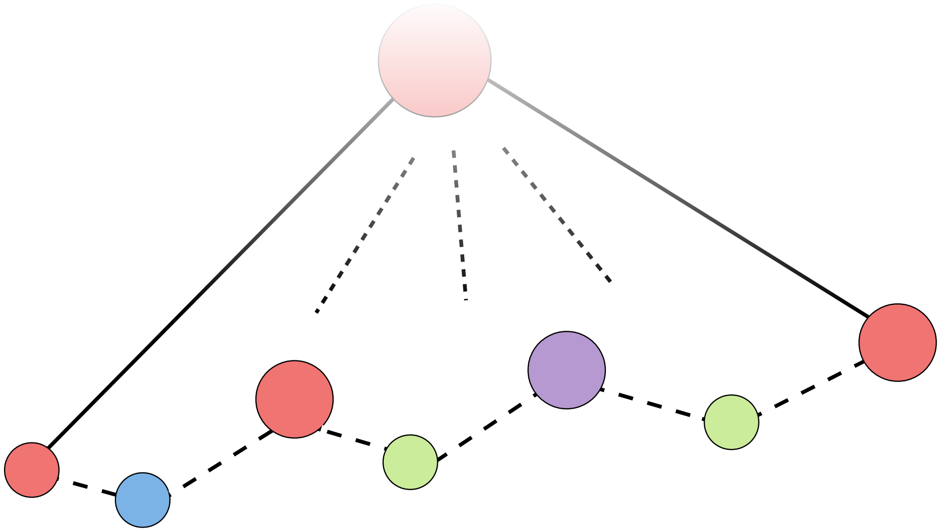
How big should a task be?

Multiple answers

- Depends on available platform parallelism
- Depends on available application parallelism
- Depends on application phases

Automatically adapt?

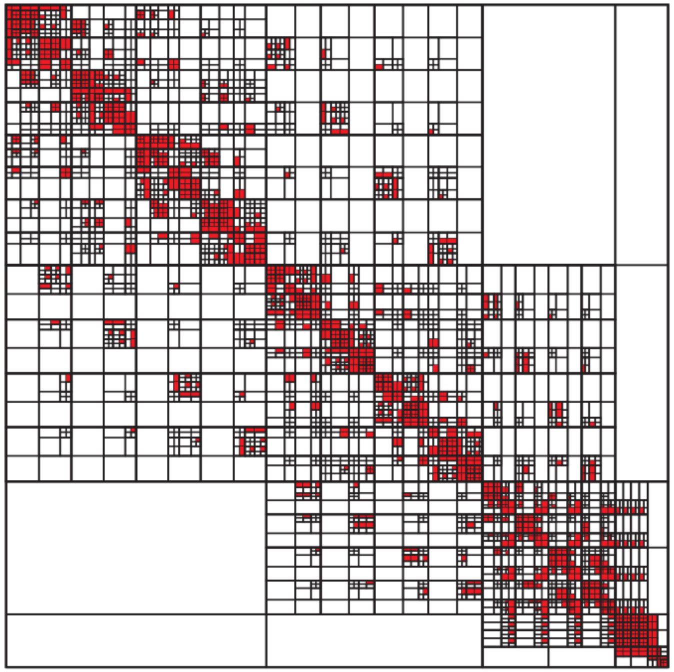
Recursive task graphs



Recursive task graphs

Applications themselves are recursive

- e.g. h-matrices

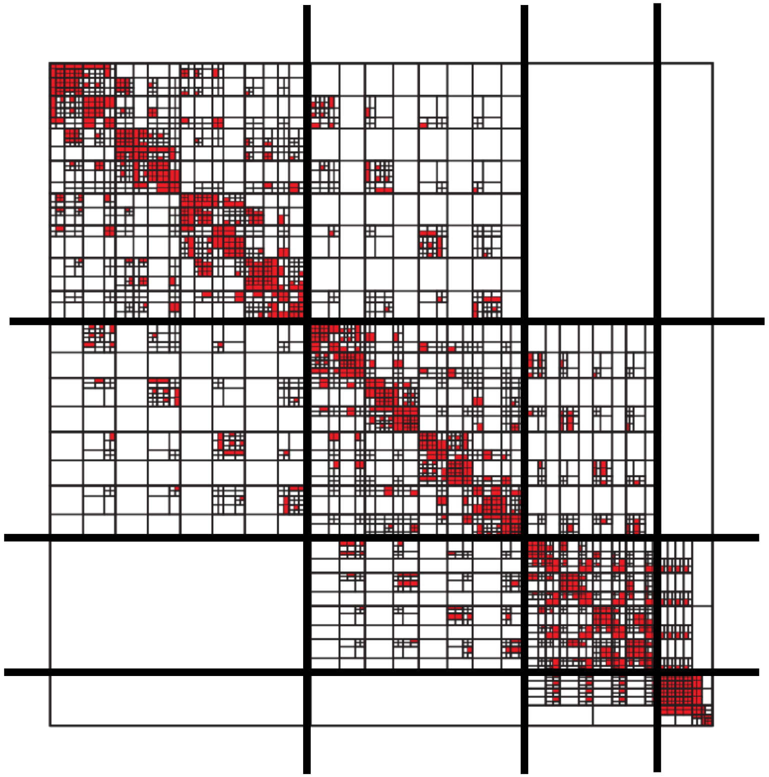


From Airbus Group

Recursive task graphs

Applications themselves are recursive

- e.g. h-matrices

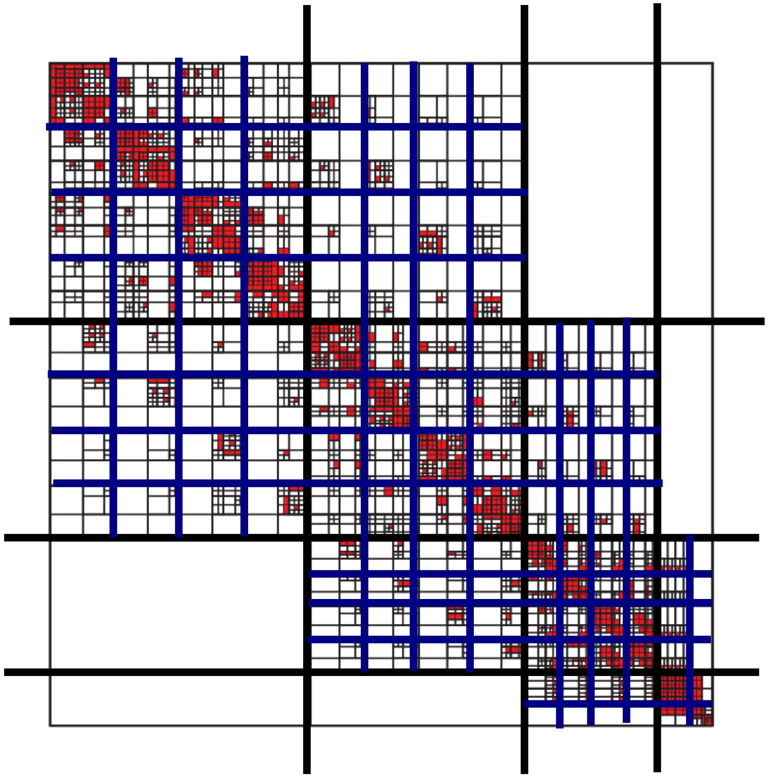


From Airbus Group

Recursive task graphs

Applications themselves are recursive

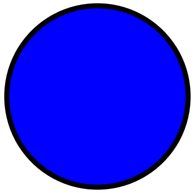
- e.g. h-matrices



From Airbus Group

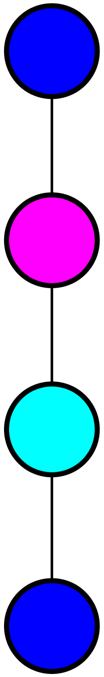
Recursive task graphs

Ideally, should just start with one huge task, and split



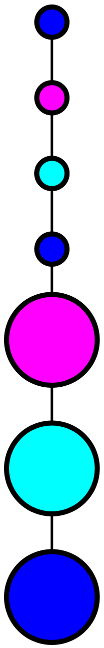
Recursive task graphs

Ideally, should just start with one huge task, and split



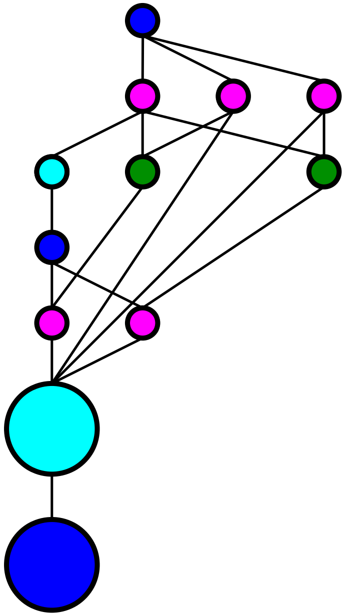
Recursive task graphs

Ideally, should just start with one huge task, and split



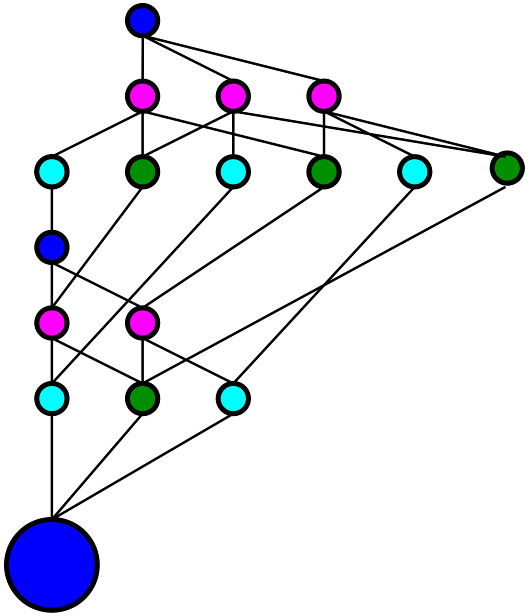
Recursive task graphs

Ideally, should just start with one huge task, and split



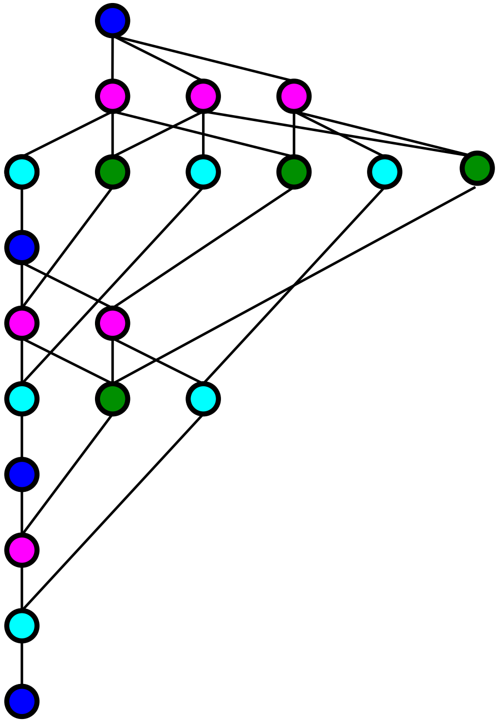
Recursive task graphs

Ideally, should just start with one huge task, and split



Recursive task graphs

Ideally, should just start with one huge task, and split



Dividing tasks

No extra synchronization

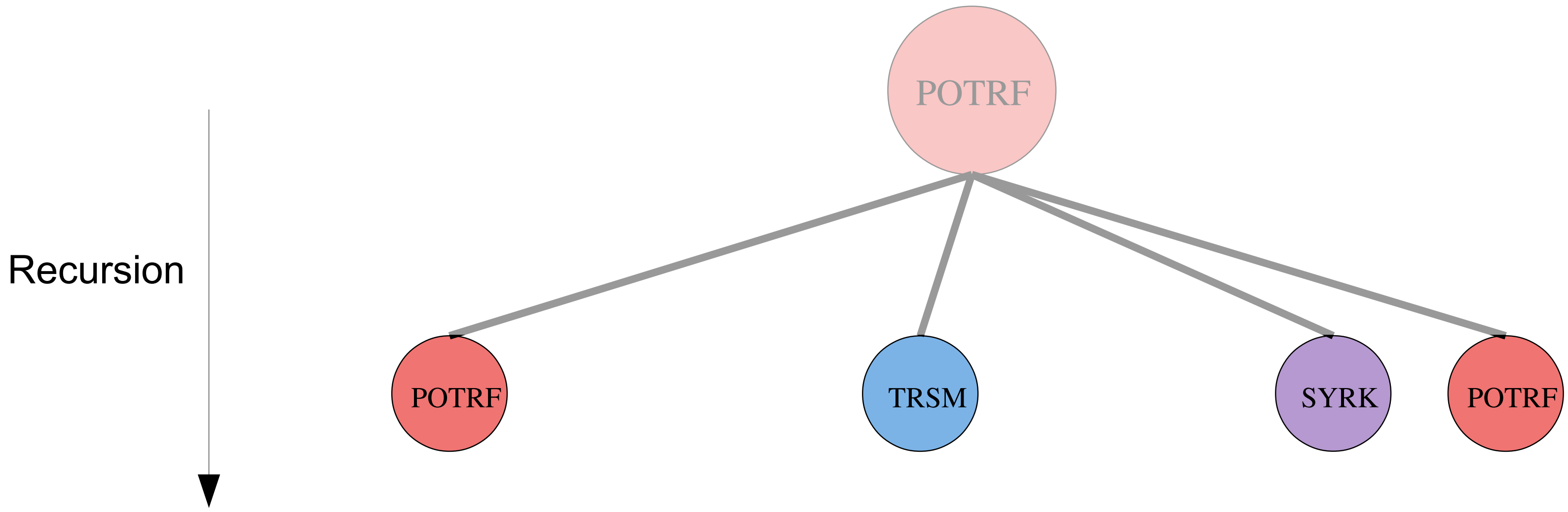
→ Can consider task graph subdivision as a tree

POTRF
Recursion

Dividing tasks

No extra synchronization

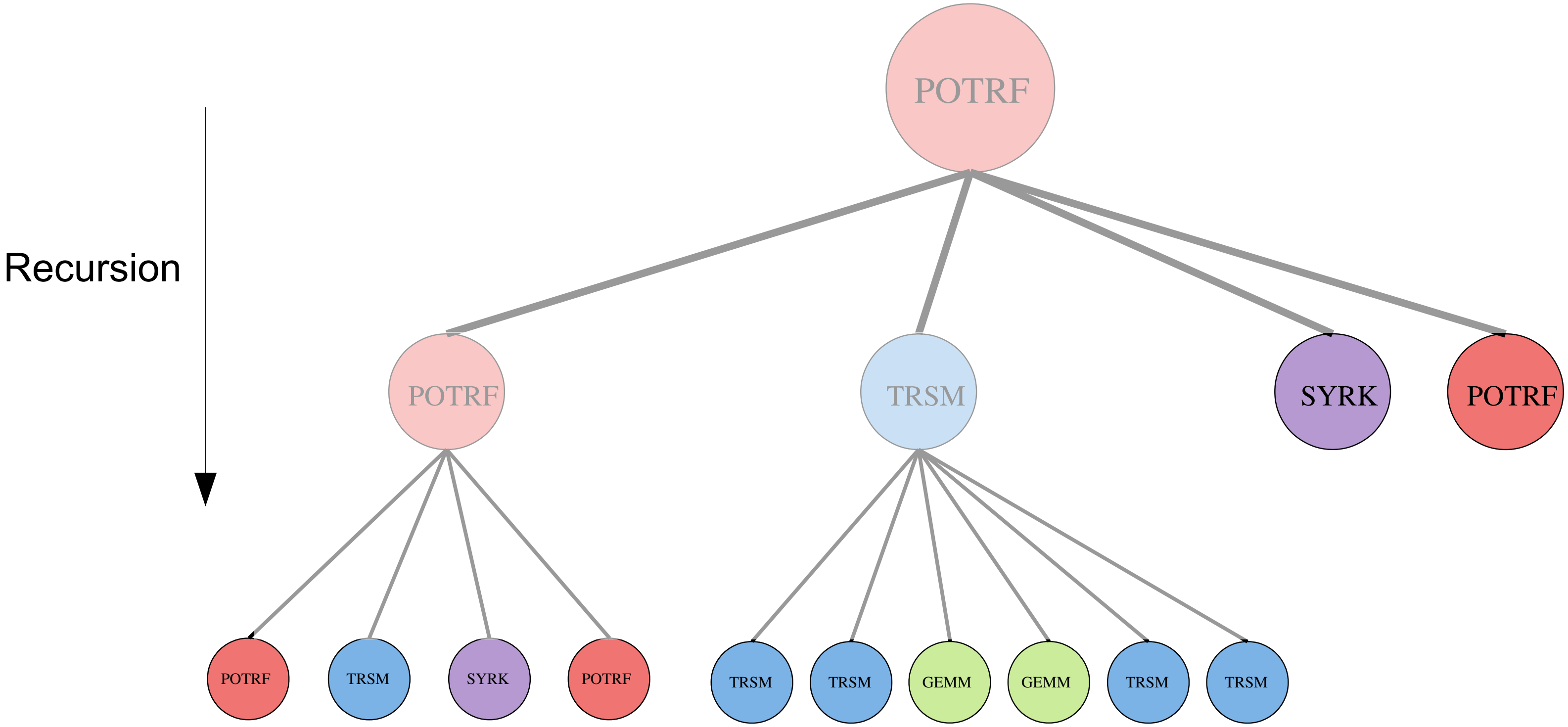
→ Can consider task graph subdivision as a tree



Dividing tasks

No extra synchronization

→ Can consider task graph subdivision as a tree

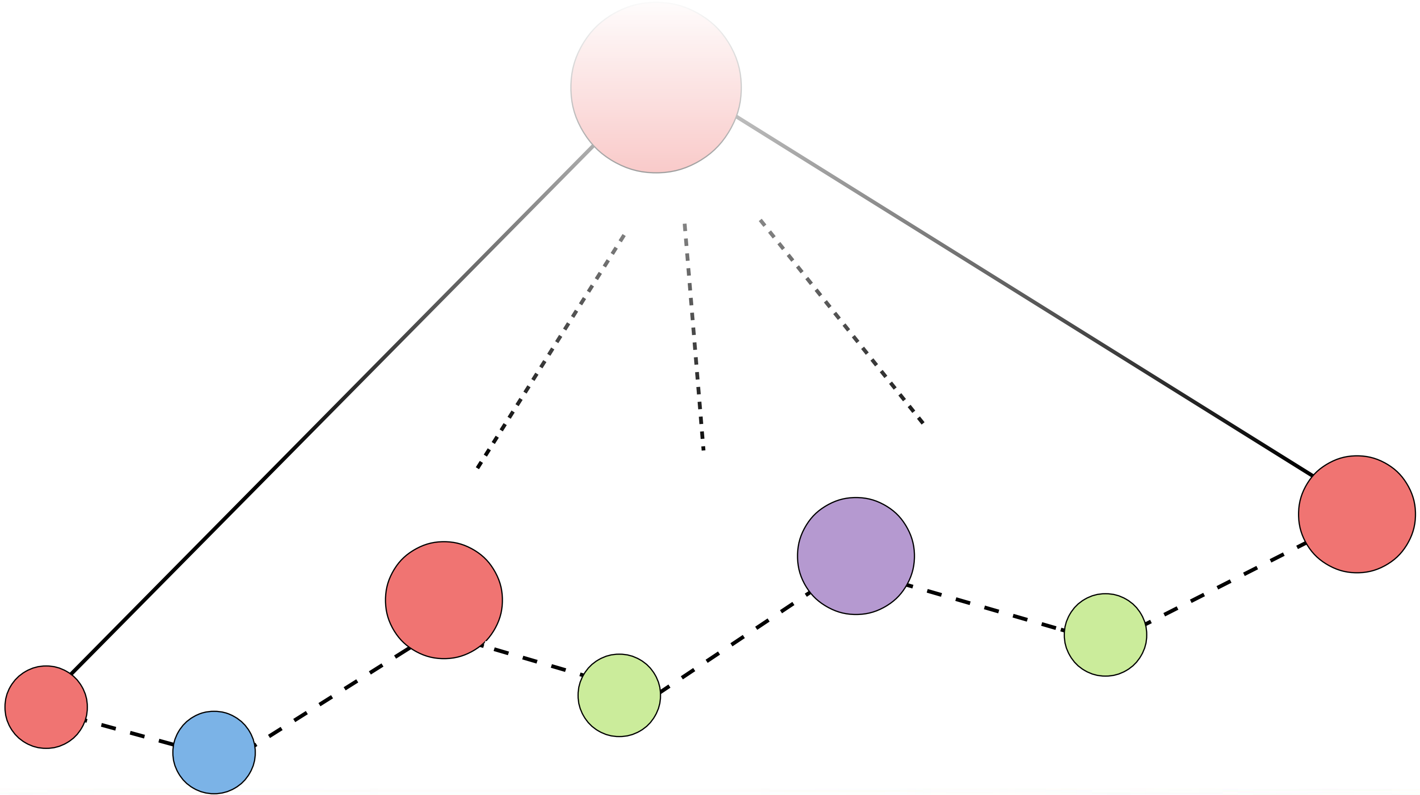


Dividing tasks

No extra synchronization

→ Can consider task graph subdivision as a tree

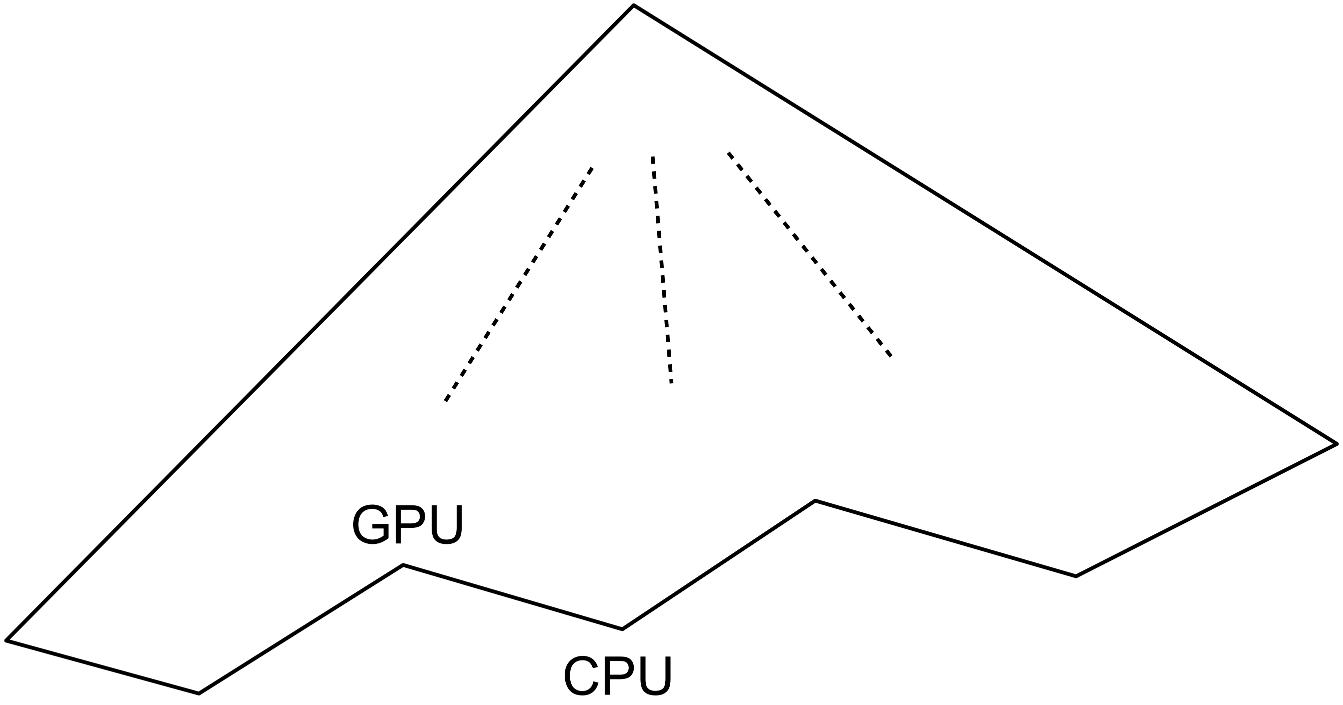
→ Decide at will where and when to stop recursing



Opportunities

GPU / CPU efficiency management

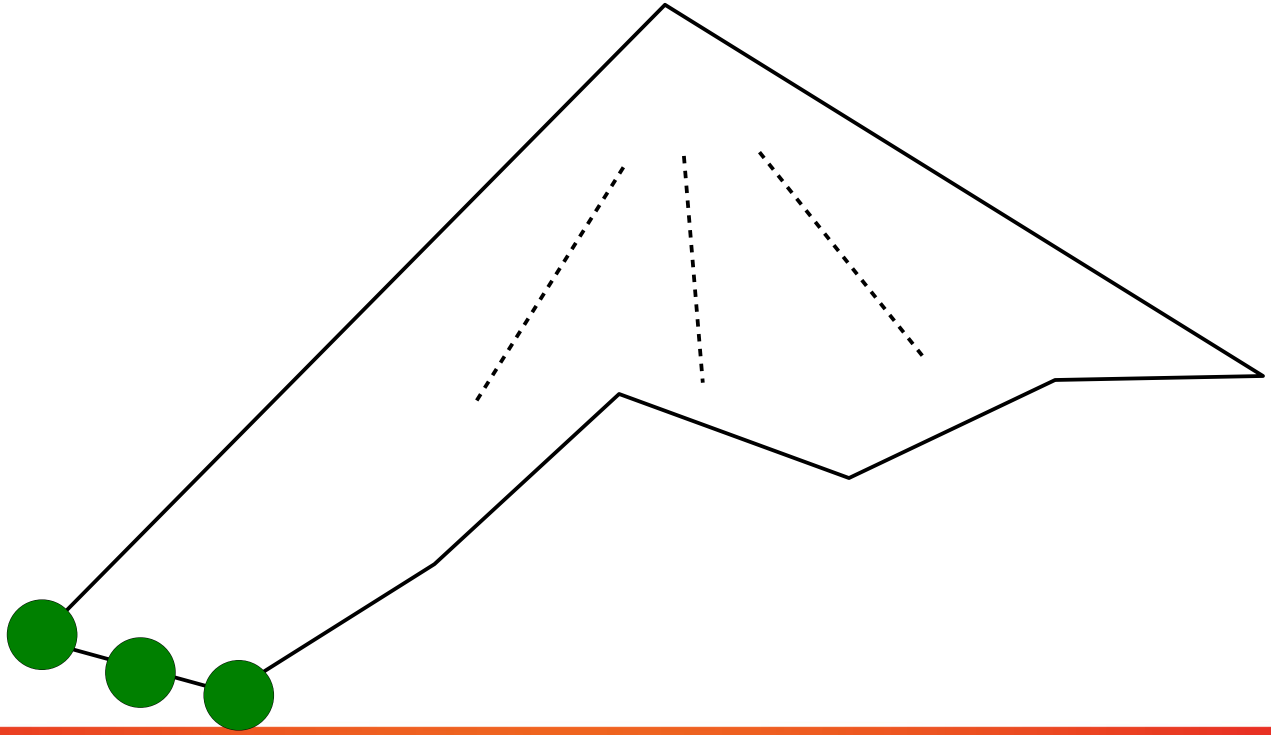
- Stop recursing at desired tile size
  - Keep large tasks for GPUs, small tasks for CPUs
- Care for latency of the task-graph critical path



Opportunities

Delay unrolling

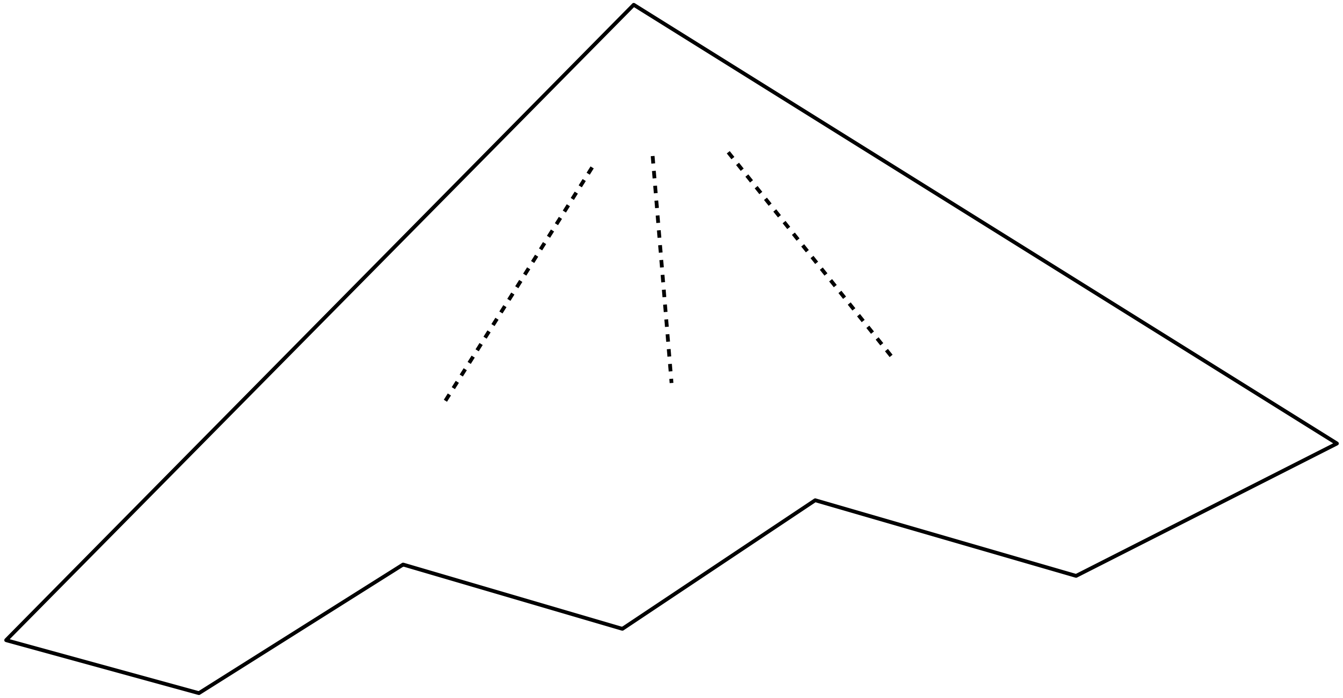
- Observe behavior before unrolling the rest accordingly
- Decorrelate submission and execution



Opportunities

Delay unrolling

- Observe behavior before unrolling the rest accordingly
- Decorrelate submission and execution



Recursive task graphs

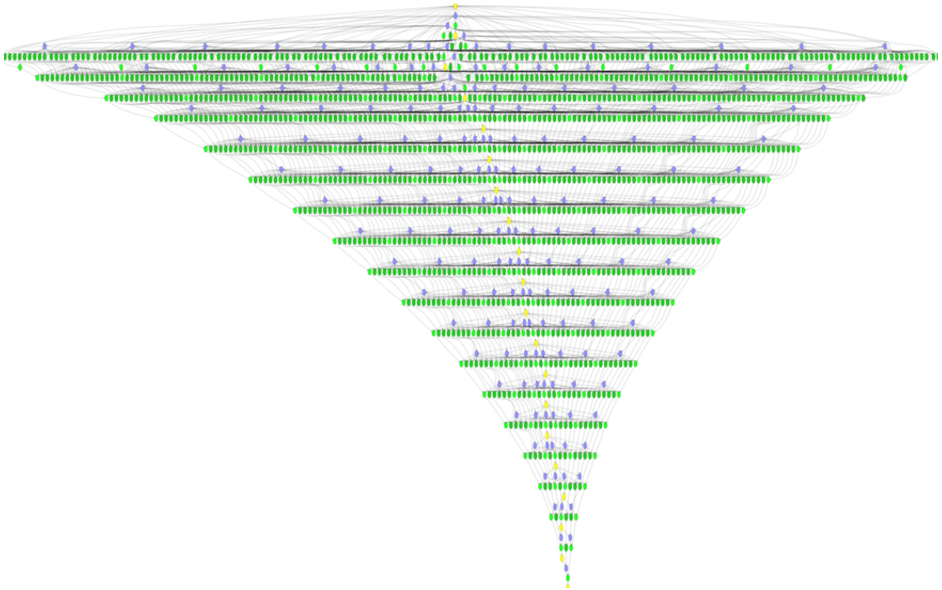
[Lucas'23]

- Made Recursive task graphs work
  - With STF etc.
  - With no spurious dependency
- Ideally, Cholesky boils down to
  - `int main(void) { potrf(A); }`
  - And the runtime splits into tasks recursively
- Now : When? How many? How? Which?

Recursive task graphs

When?

- Better wait a bit to see how things behave?
- In Cholesky case
  - Beginning: better not split too much
    → Efficiency
  - End: better split more
    → Parallelism



Recursive task graphs

How many?

- GPUs? Better not split too much
- CPUs? Better split
- Priority? Better split
- ...

Recursive task graphs

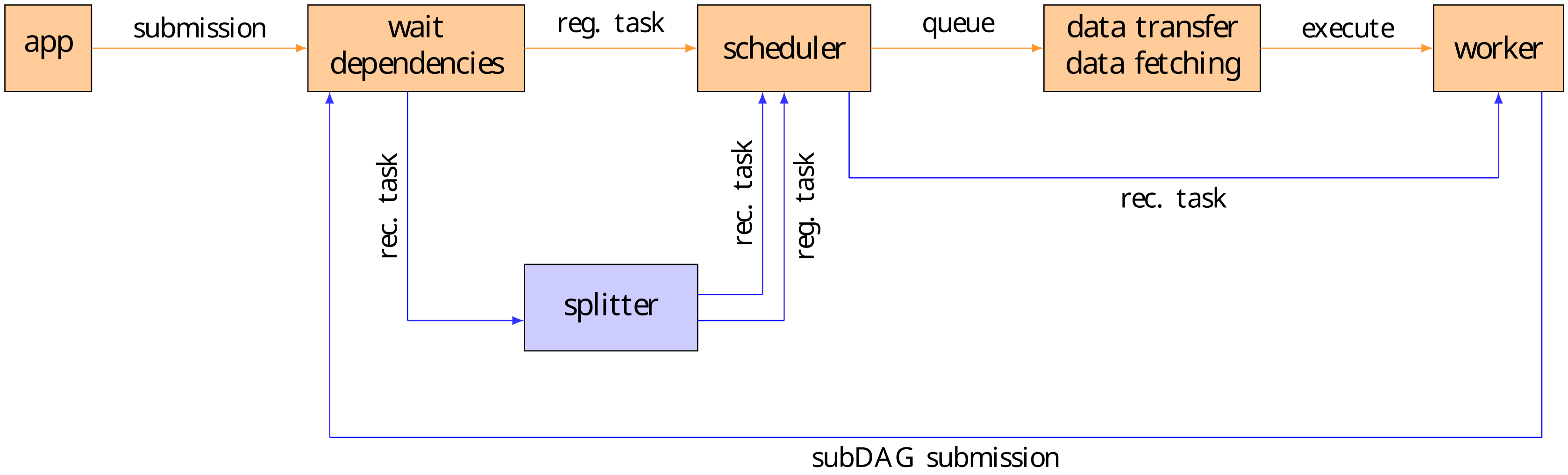
Which?

- Depends on split efficiency
  - Splitting may lower GFlop/s achieved by tasks
  - But also exposes slack
    - E.g. mix POTRF subtasks with other subtasks
- Depends on priority
  - Better split tasks on the critical path

    → Reduce latency

Recursive task graphs

When: at which scheduling stage?



- Runtime information
- Still parallel submission

Complex problem

Complex problem

I’m a system guy

Complex problem

I’m a system guy

- Let’s try simple solutions, at runtime

Complex problem

I’m a system guy

- Let’s try simple solutions, at runtime
- Hoping to inspire you into complex solutions

3 simple solutions presented here

First results : heterogeneous

[Morin'25]

Consider acceleration and parallelism availability

- Big tiles for GPUs
- Small tiles for CPUs
- Find a balance between the two

First results : heterogeneous

[Morin'25]

Consider acceleration and parallelism availability

- Big tiles for GPUs
- Small tiles for CPUs
- Find a balance between the two

Run a linear program every 50 tasks (takes < 1ms)

- Optimizes splitting ratios per task type and recursion level
- Ratios serve as splitting guide

First results : heterogeneous

Parameters

$\mathcal{T}, N_{t,l}^{tot}$	$t \in \mathcal{T}$	Set of task types, Number of task not split of type t at level l.
$\mathcal{R}, R^u$	$u \in \mathcal{R}$	Set of processing unit types, Number of PU of type u.
$\mathcal{L}$		Maximum level of recursion.
$MinN_u$	$u \in \mathcal{R}$	Minimal wanted number of tasks on each PU of type u

Variables

exT		Total execution time.
Ns_l^t	$t \in \mathcal{T}, l < \mathcal{L}$	Number of split task of type t and level l.
$Ne_{l,u}^t$	$t \in \mathcal{T}, l \leq \mathcal{L}, u \in \mathcal{R}$	Number of task of type t and level l executed on PU u

First results : heterogeneous

Minimize exT

Subject to

Task number splitting.

$$\sum_{u\in\mathcal{R}} Ne^{t}_{l,u} + Ns^{t}_{l} - \sum_{p\ \in\ par(t)} nch^{t}_{p,l}\cdot Ns^{p}_{l-1} \geq N^{tot}_{t,l} \qquad t\in\mathcal{T},\ l\leq\mathcal{L}$$

No last−level splitting.

$$Ns^{t}_{\mathcal{L}} = 0 \qquad \forall\ t\in\mathcal{T}$$

Completion time when executing tasks.

$$\sum_{\substack{t\in\mathcal{T}\\ 0\leq l\leq\mathcal{L}}} Ne^{t}_{l,u}\cdot Ex^{u}_{t,l} - R^{u}\cdot exT \leq 0 \qquad u\in\mathcal{R}$$

Minimal number of tasks on PU type.

$$\sum_{\substack{t\in\mathcal{T}\\ 0\leq l\leq\mathcal{L}}} Ne^{t}_{l,u} - R^{u}\cdot MinN_{u} \leq 0 \qquad u\in\mathcal{R}$$

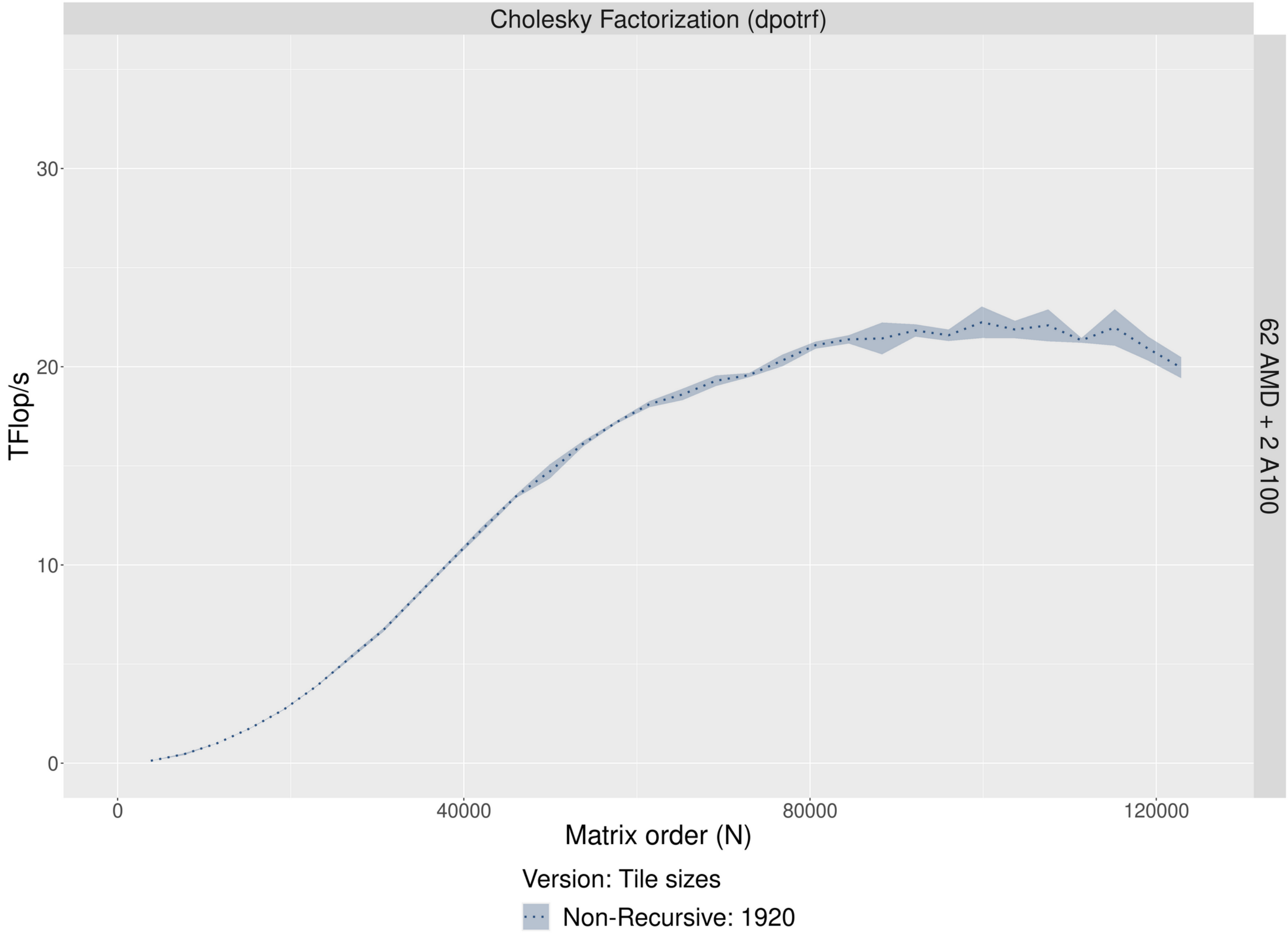
First results : heterogeneous

Cholesky factorization

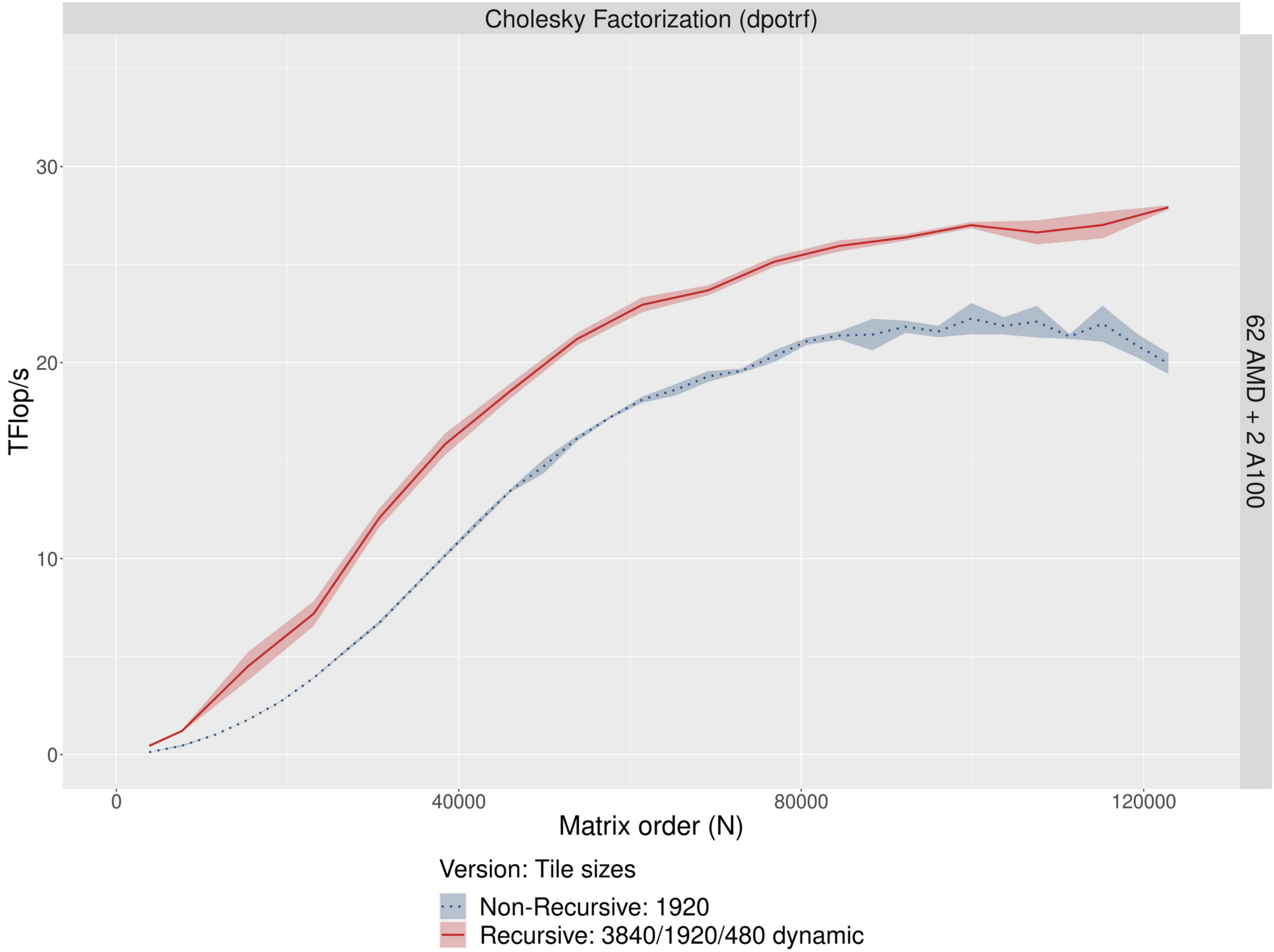
- 2x32 core AMD Zen3 EPYC 7513 @ 2.6GHz
- 2x NVIDIA A100
- Scheduler : Deque Model Data-Aware Ready (DMDAR, ≃HEFT)

- Tile sizes
- « big » : 3840, the most efficient
- « small » : 480, parallelism, for CPU cores
- « medium » : 1920, 2560, trade-off

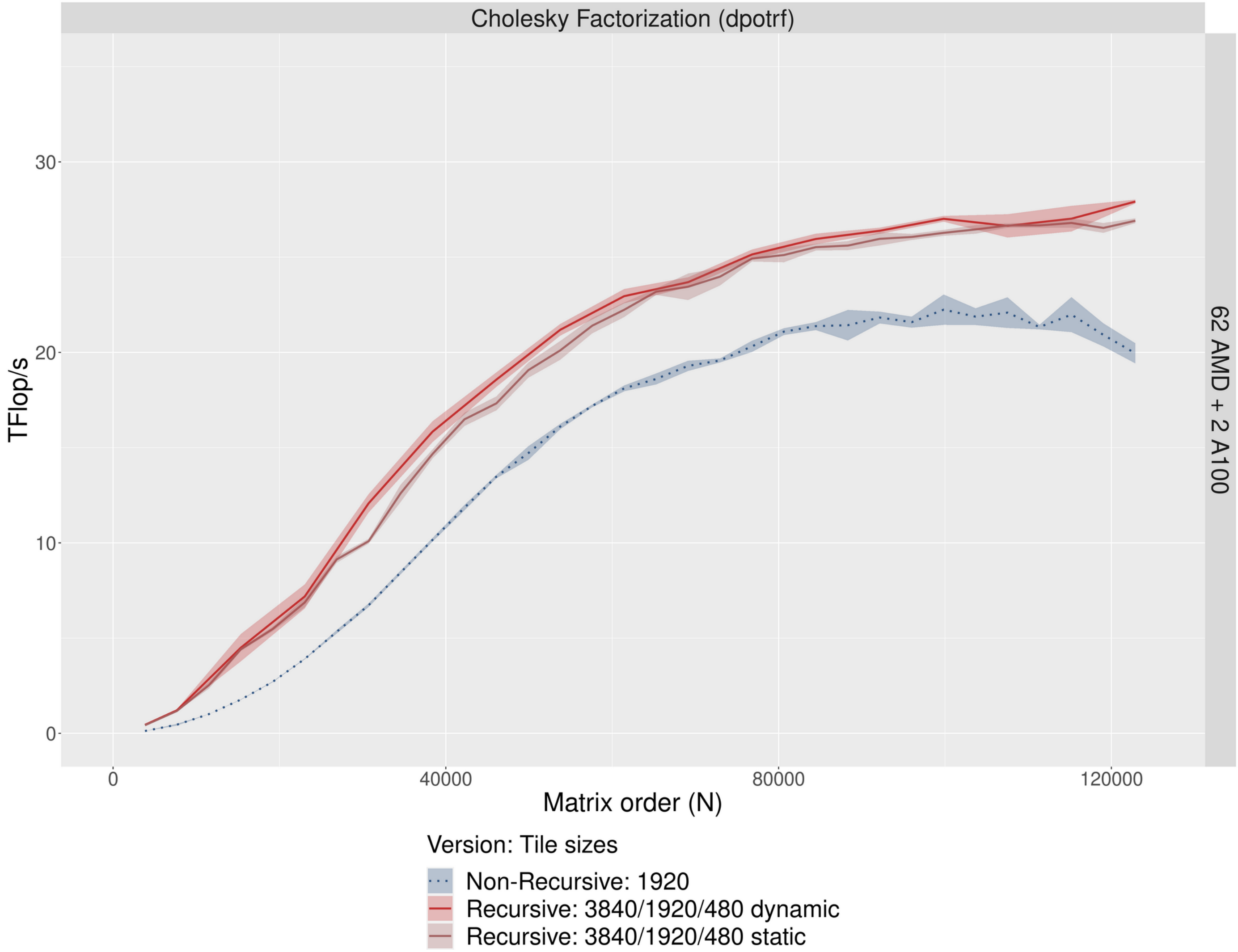
First results : heterogeneous Cholesky



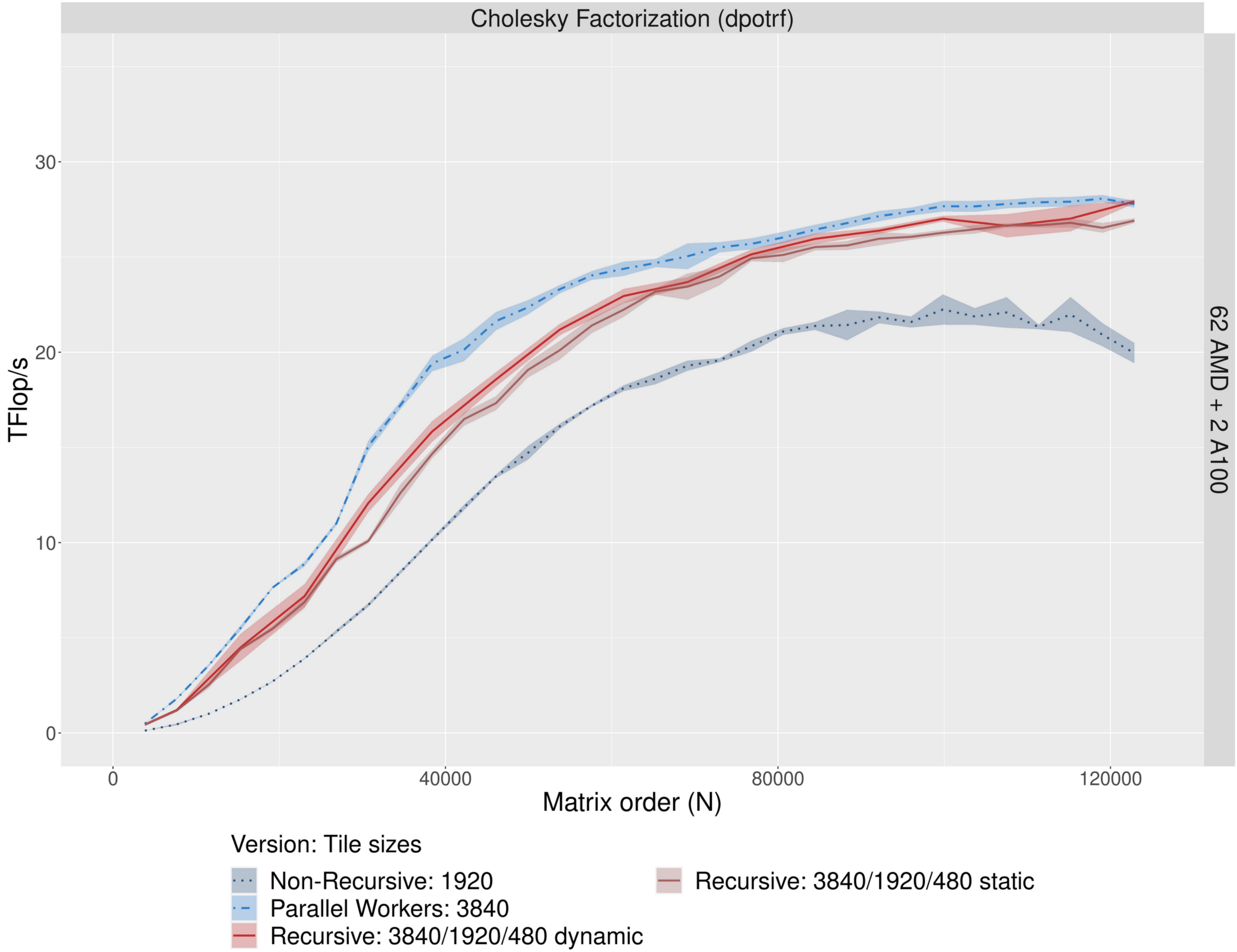
First results : heterogeneous Cholesky



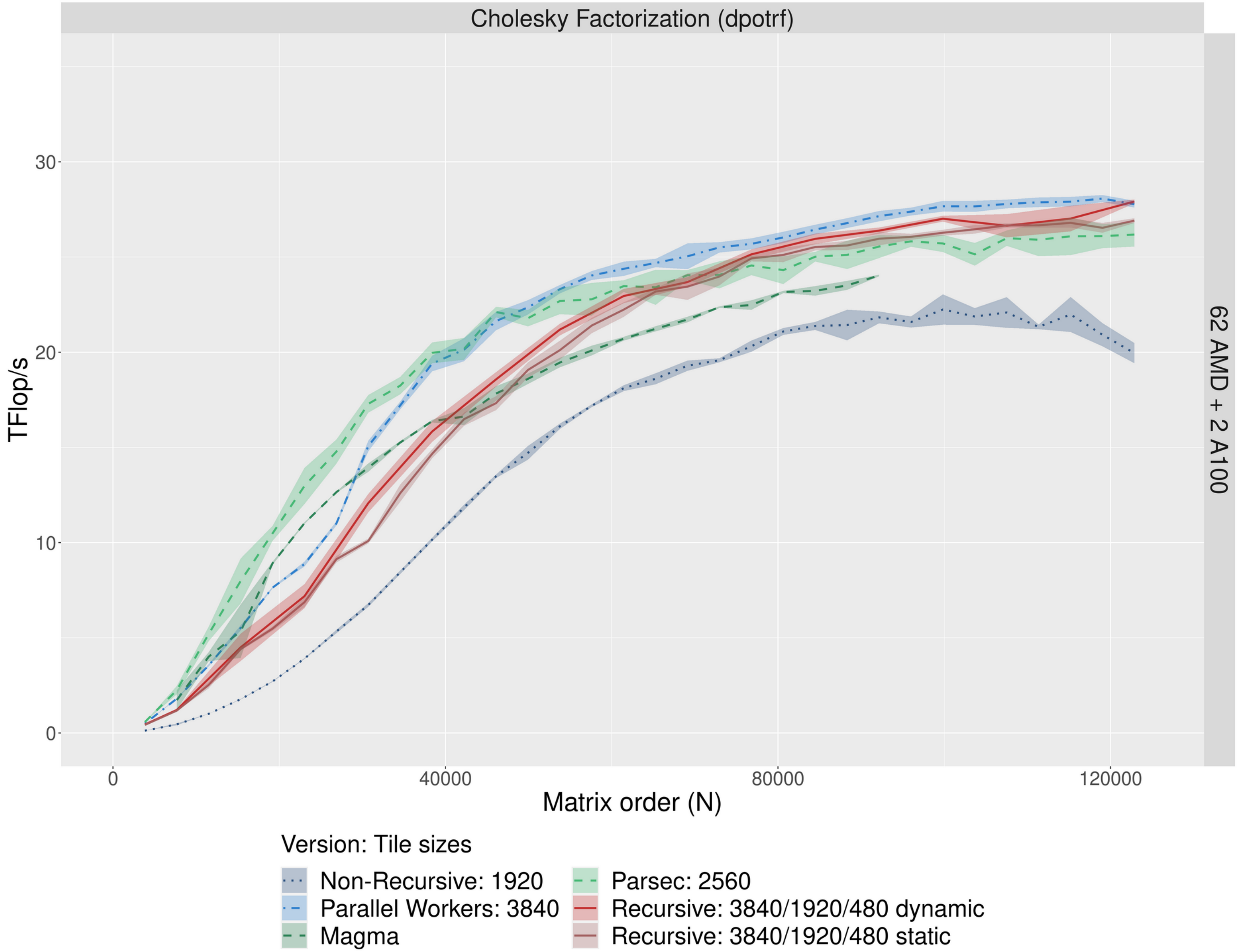
First results : heterogeneous Cholesky



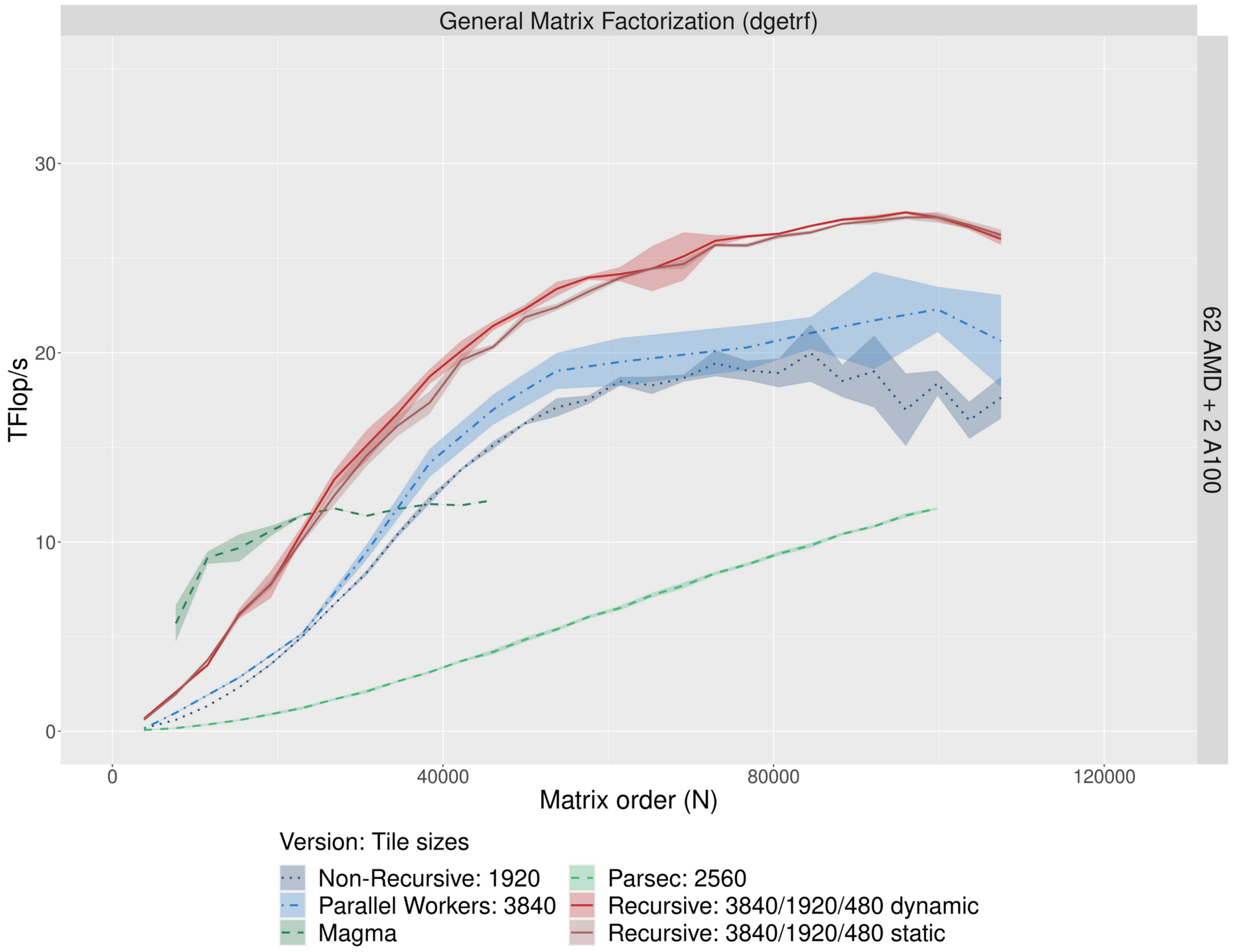
First results : heterogeneous Cholesky



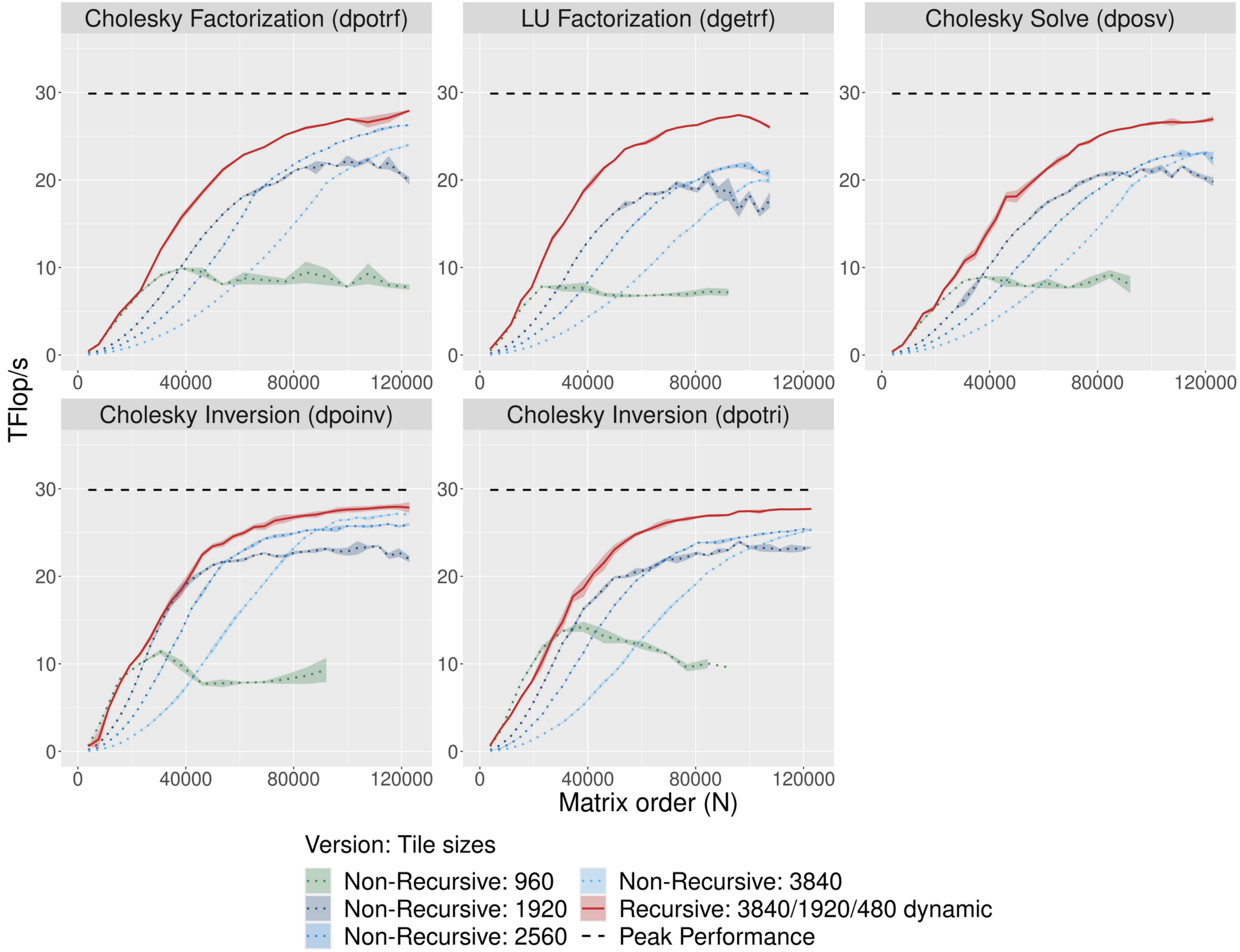
First results : heterogeneous Cholesky



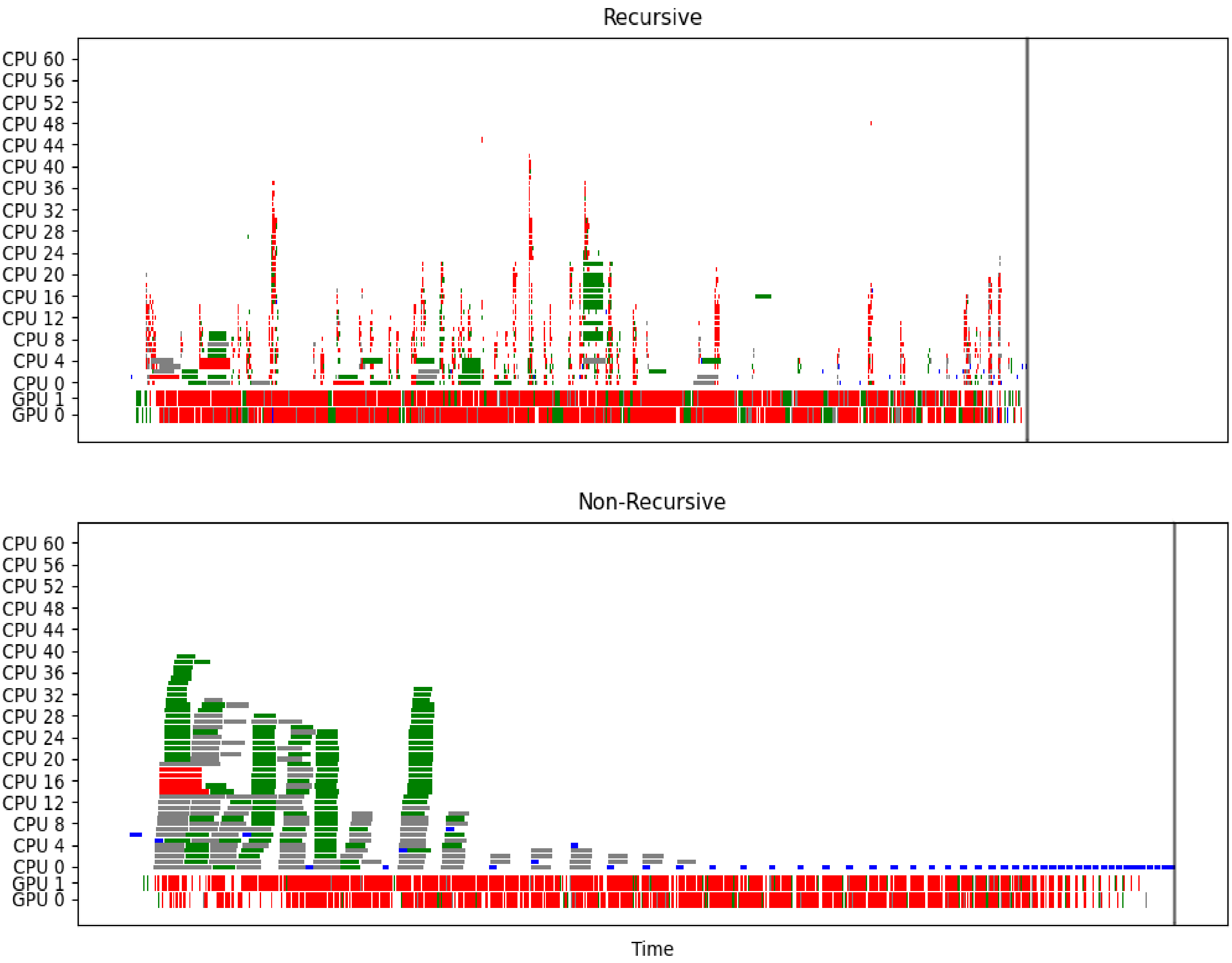
First results : heterogeneous LU



First results : heterogeneous



First results : heterogeneous



First results : heterogeneous

Linear Program

- Generic, automatic
- Requires performance models

but

- Not very reactive to new situation
- Doesn't take data transfers into account
- Global view, hard to turn into local action
  - Splitting ratios
  - Sometimes uses too much CPU time
- Not really intuitive behavior

Mid results : more greedy approach

[Morin’26]

Why splitting?

- Decrease GPU use for non-really-GPU-efficient tasks
- Occupy CPU cores

Mid results : more greedy approach

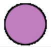
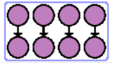
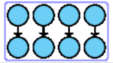
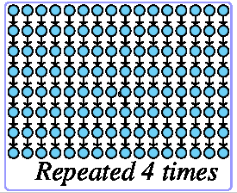
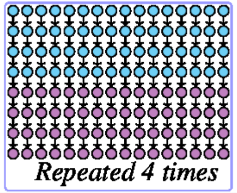
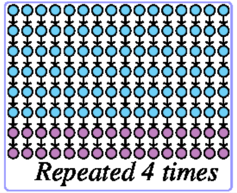
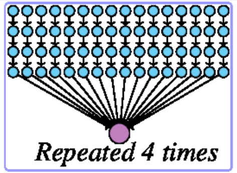
Recursive performance model : for each possible subDAG, record

- Global finish time
- Sequential GPU time
- Average # of CPU cores used

and prune symmetries

→ Splitting Profiles

Example on GEMM with 2 GPUs:

	Sub-DAG-placement (CPU / GPU)	Makespan	GPU time	Mean CPUs used (% total CPUs)
1		7 ms	7 ms	0 core (0%)
2		6 ms	12 ms	0 core (0%)
3		1088 ms	0 ms	4 cores (6%)
4		74.4 ms	0 ms	64 cores (100%)
5		43.6 ms	12.8 ms	54.7 cores (85%)
6		59 ms	6.4 ms	60.6 cores (95%)
7		40.2 ms	6 ms	59.3 cores (93%)

Mid results : more greedy approach

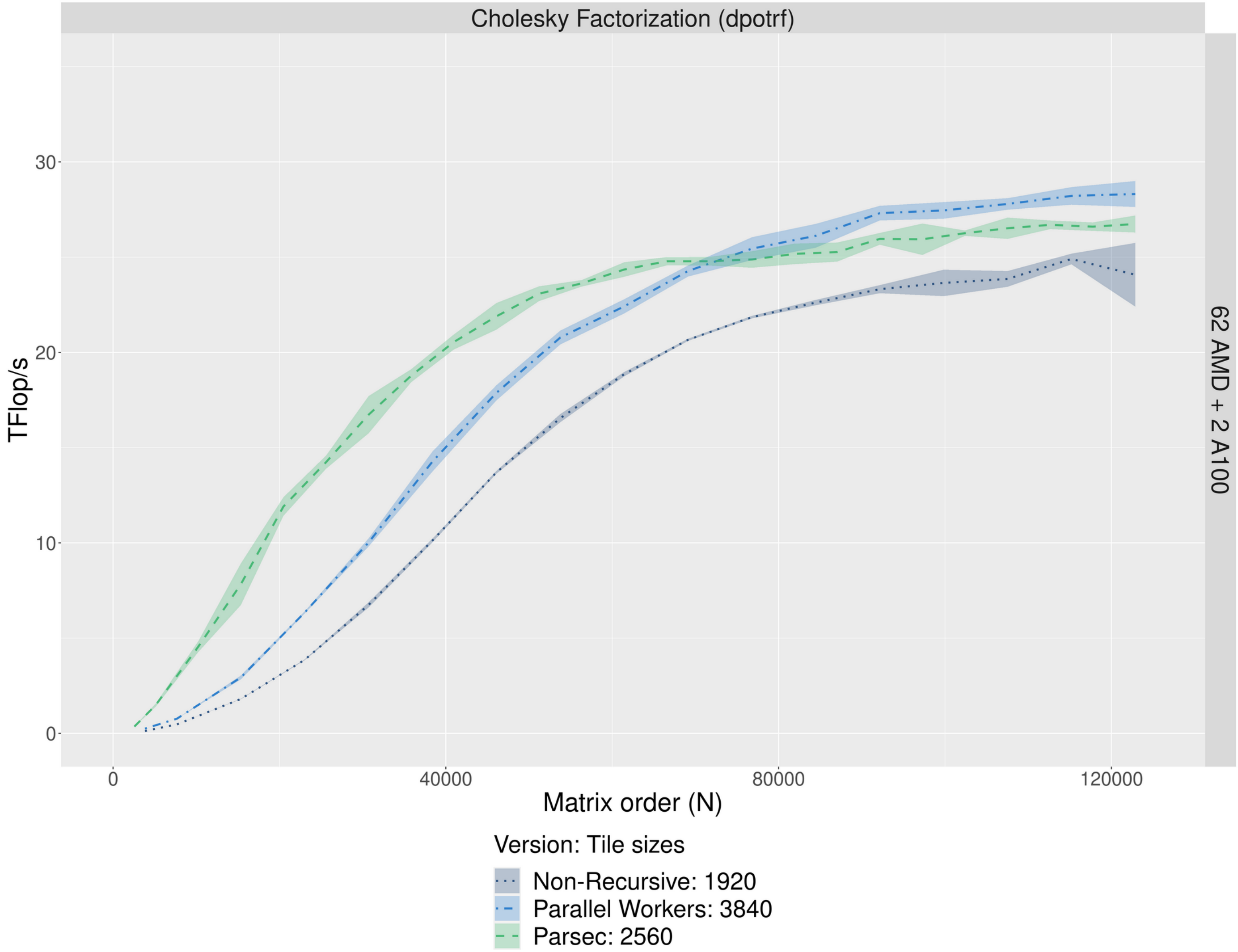
Greedy scheduler : GASPP

- Keeps ready tasks ordered by CPU/GPU acceleration
- Records how many CPUs are idle

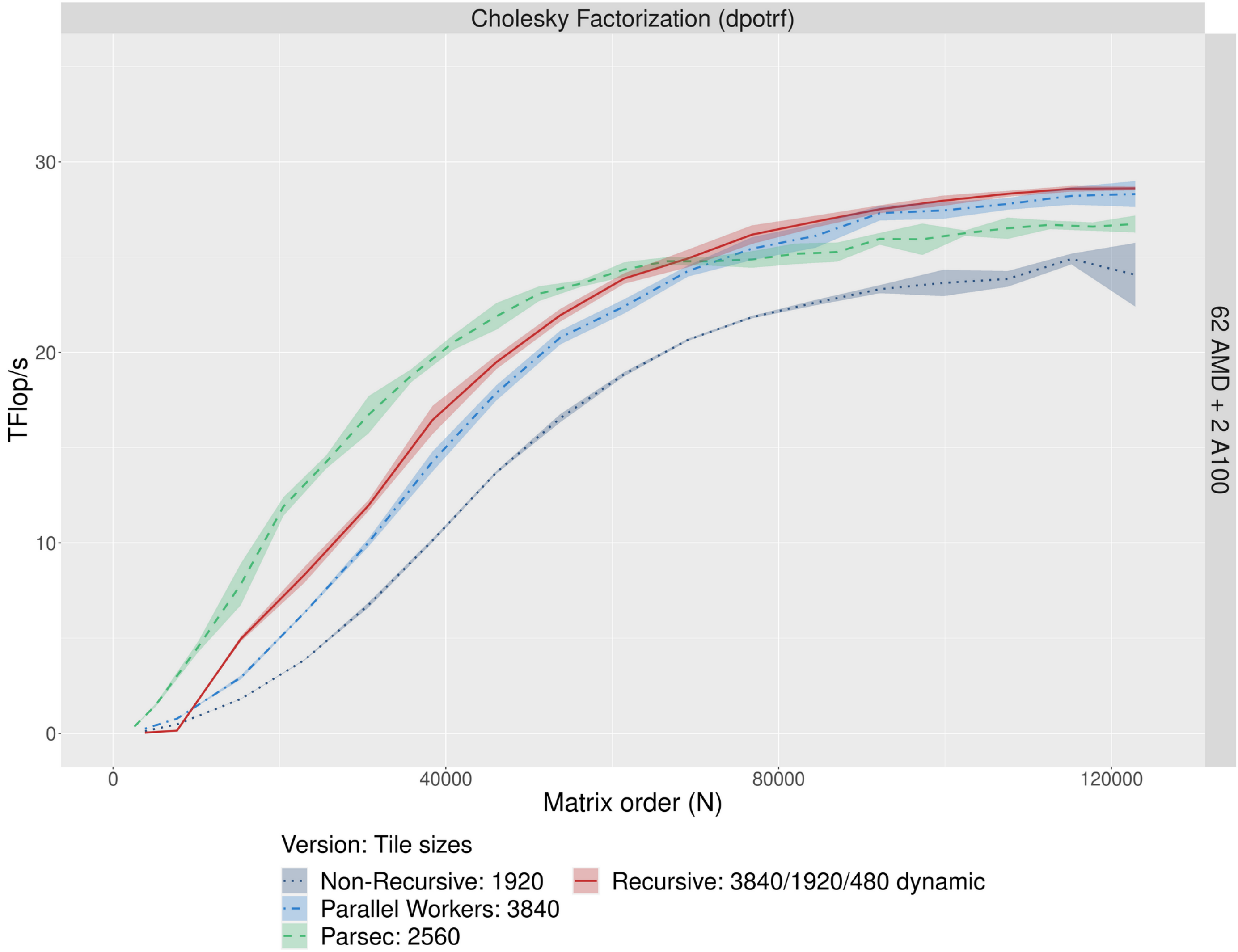
When scheduler asks for a task,

- Pick up the least-accelerated task
- Look for best splitting profile that
  - Can fit the idle CPUs
  - Allows to terminate earlier than on GPU
  - Offloads the most work from GPU
  - I.e. safe offload bet
- If none, pick up instead the most-accelerated task
  - Will go on GPU

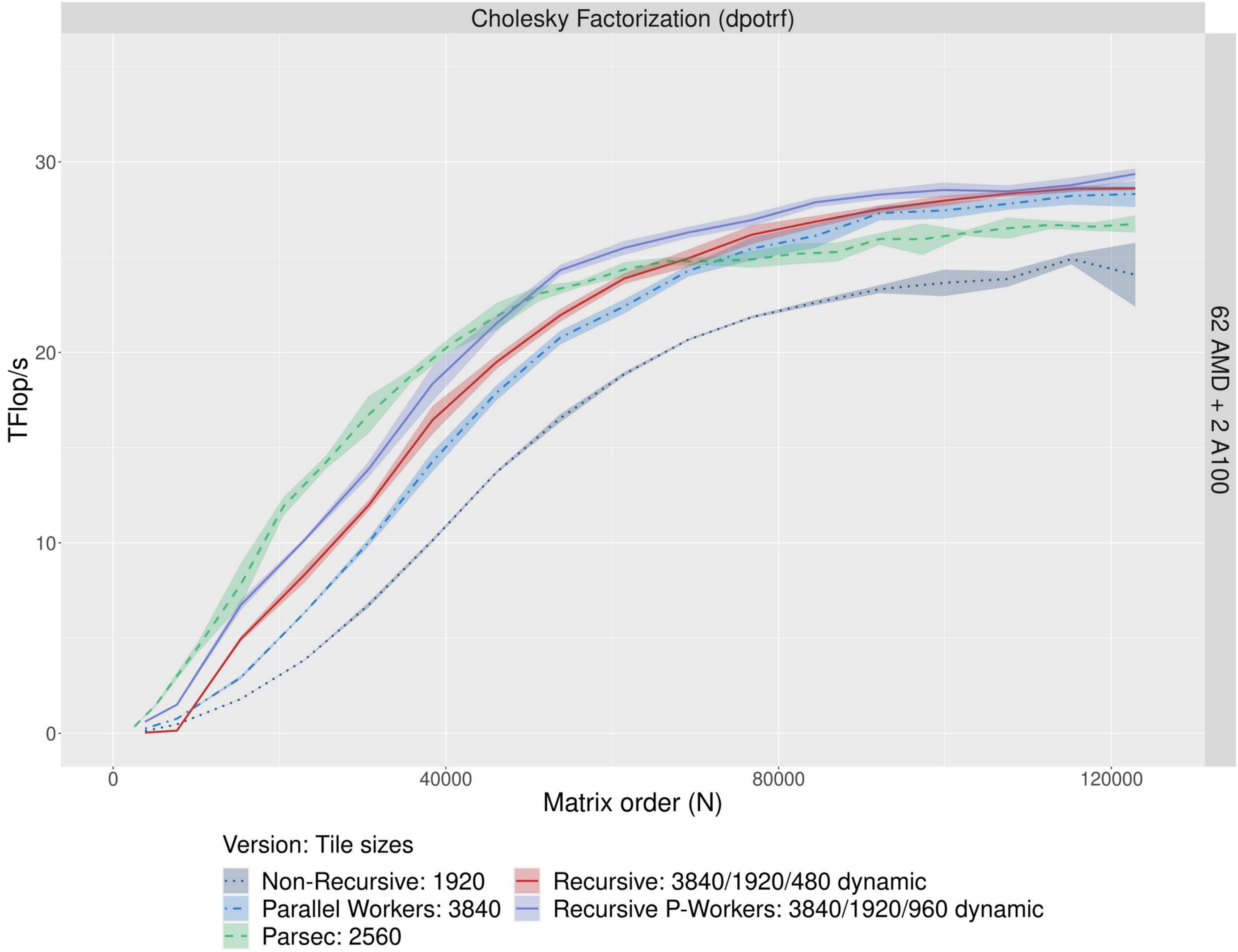
Mid results : heterogeneous Cholesky



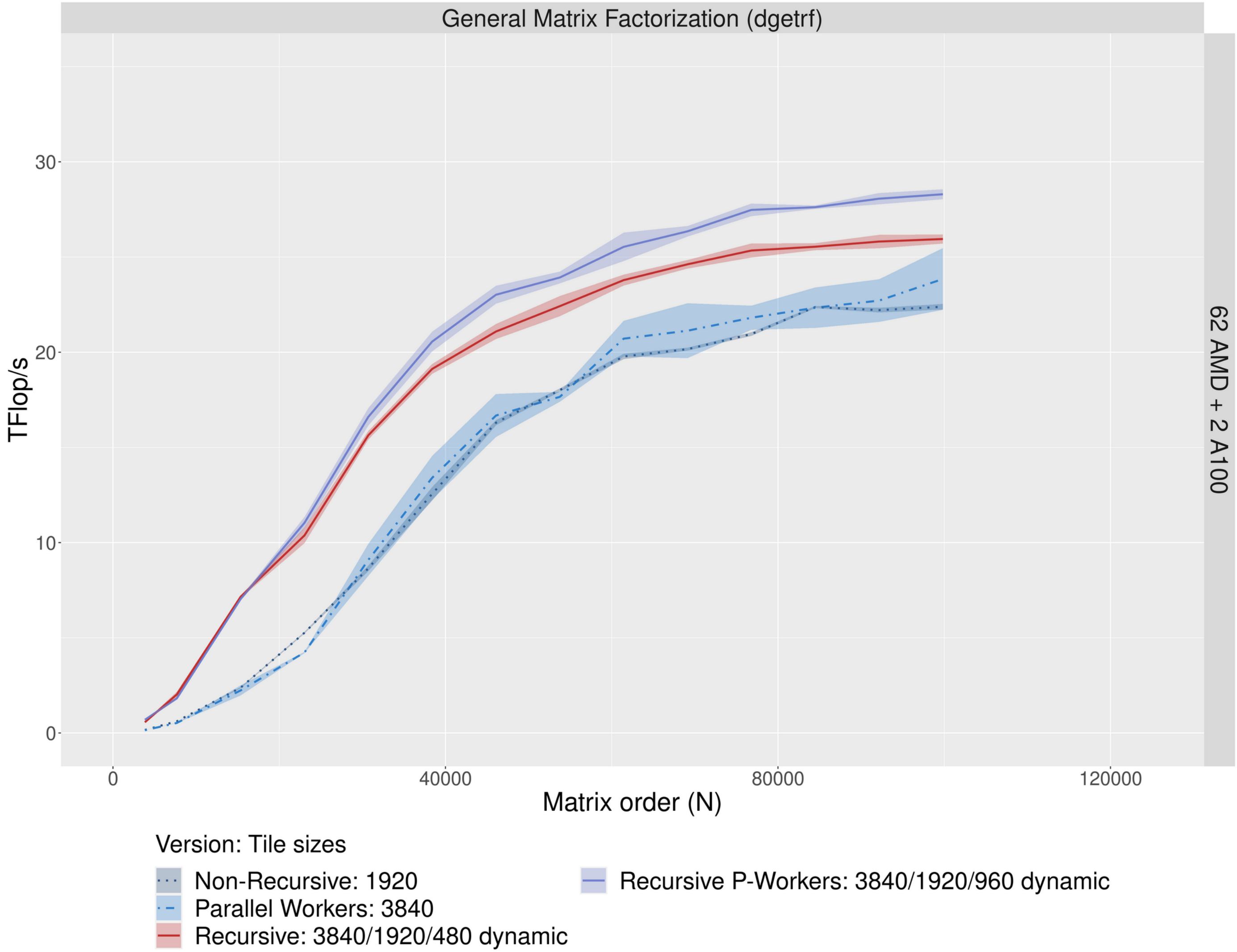
Mid results : heterogeneous Cholesky



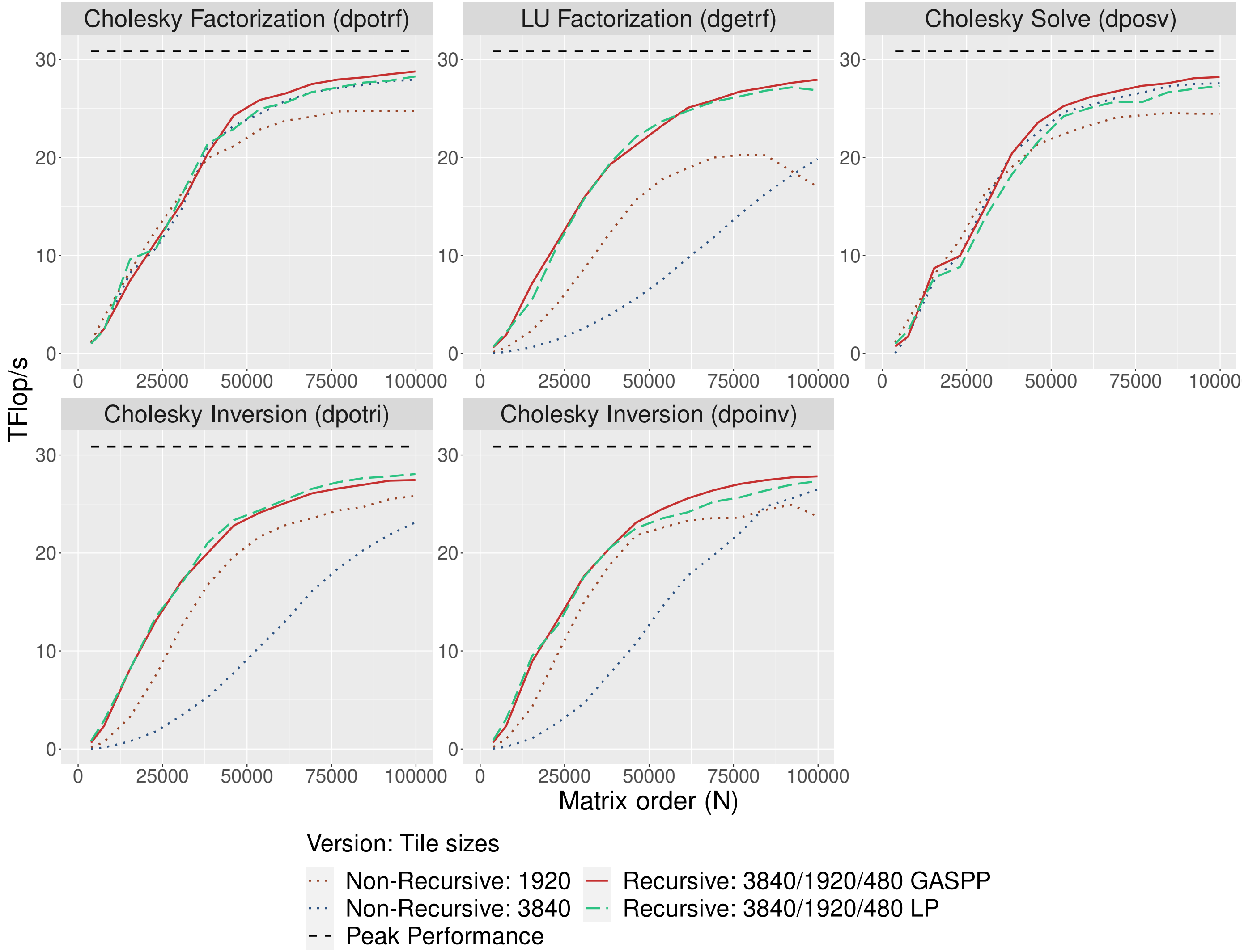
Mid results : heterogeneous Cholesky



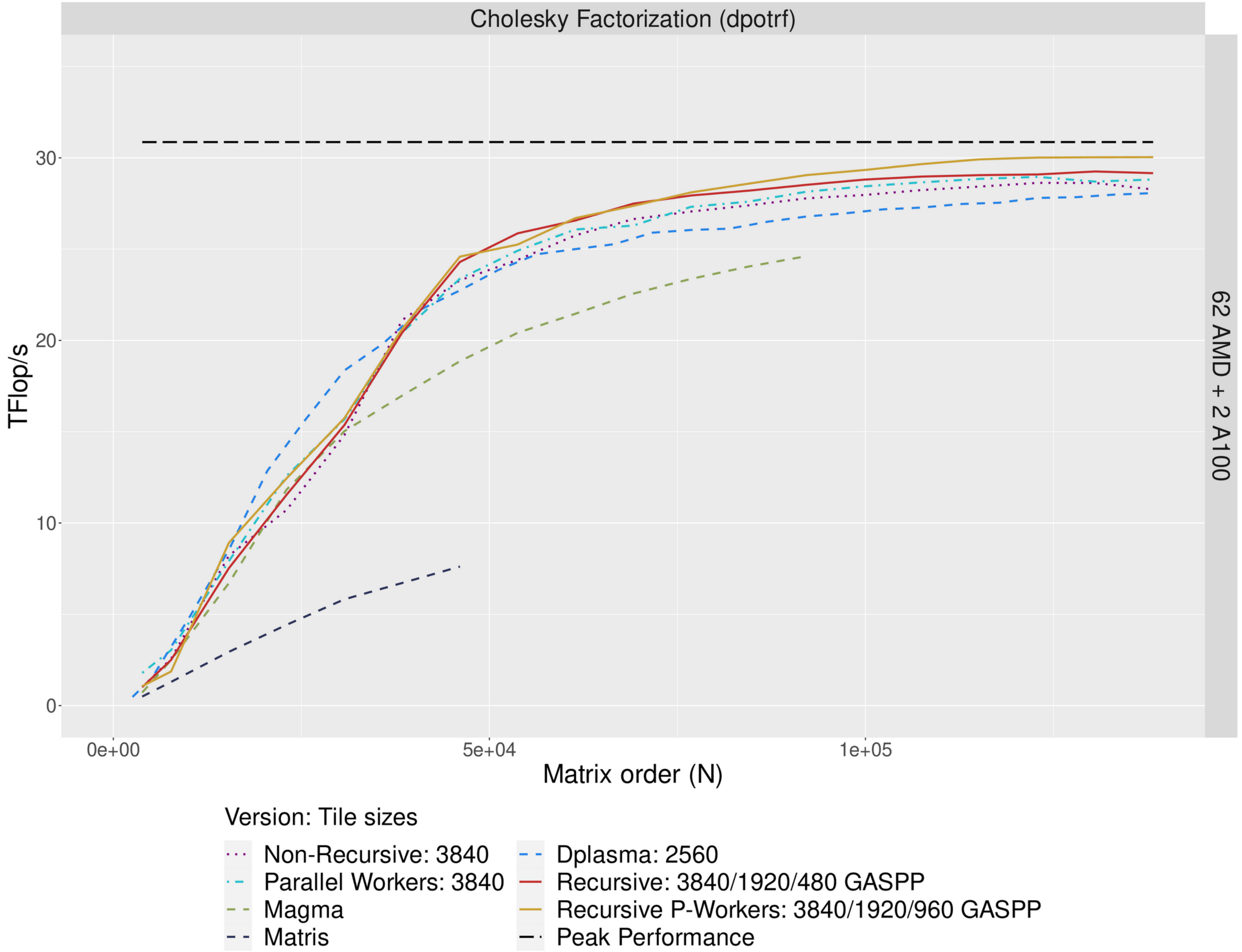
Mid results : heterogeneous LU



Mid results : heterogeneous



Mid results : heterogeneous



Current results : local/global view?

Good results asymptotically

Could be better for smaller matrices

- Not enough parallelism to occupy GPUs
- Should split a bit for GPUs too!
  - When starvation comes
  - How to evaluate starvation?

Current results : local/global view?

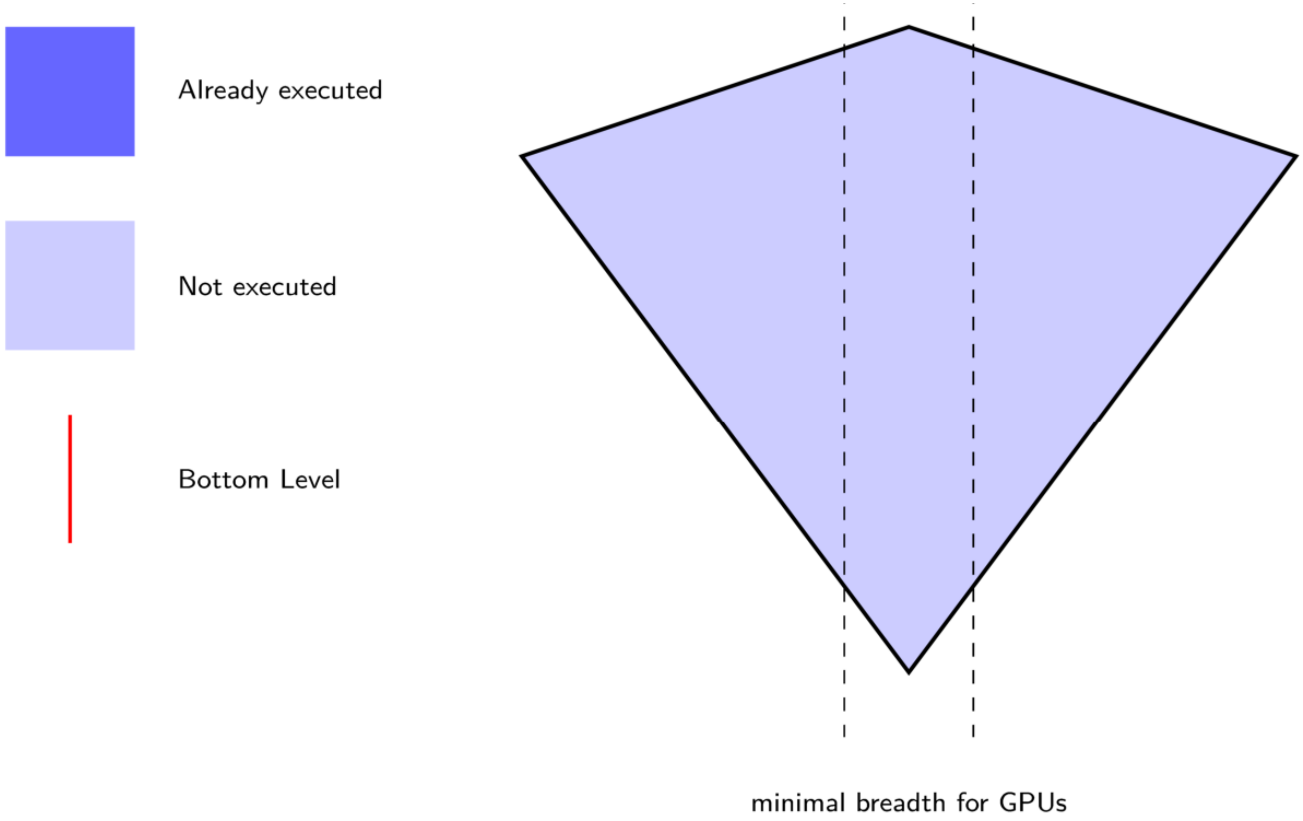
Good results asymptotically

Could be better for smaller matrices

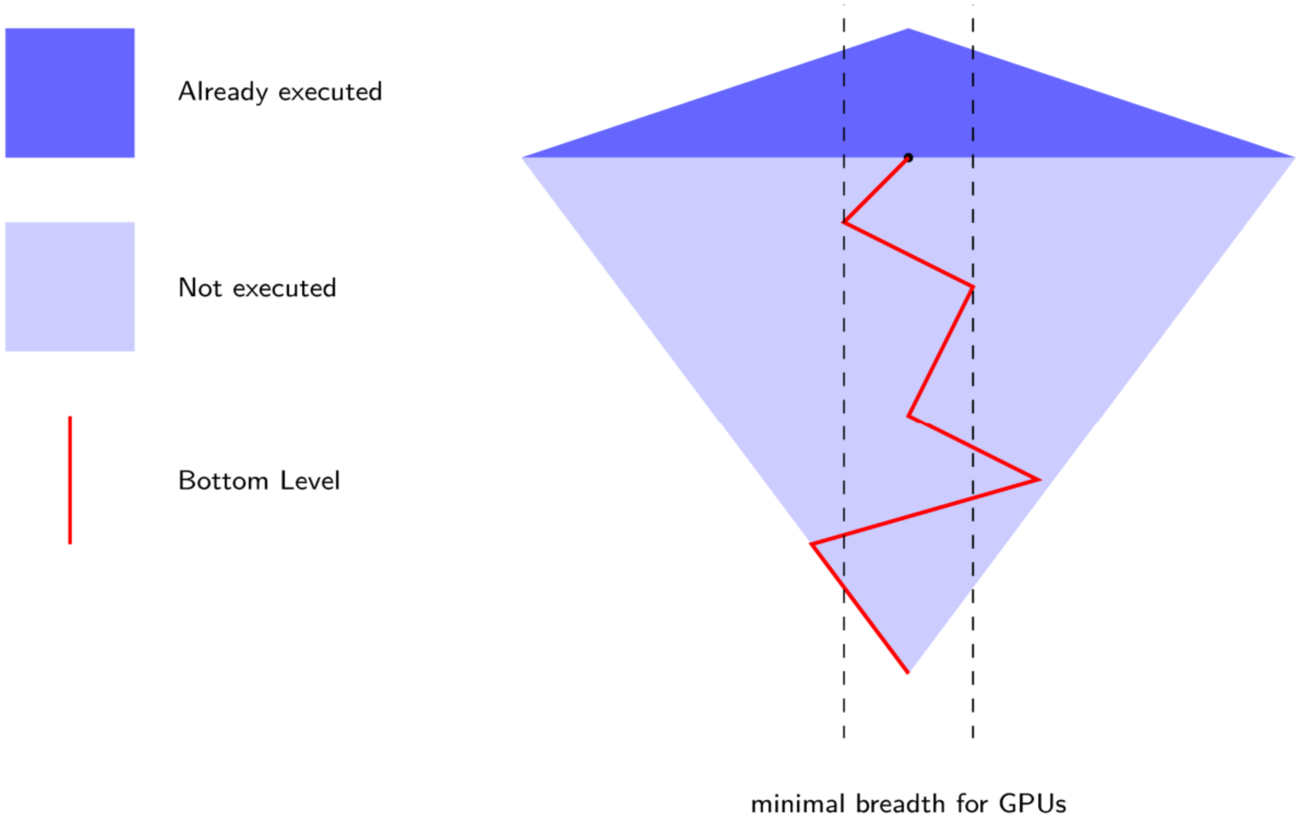
- Not enough parallelism to occupy GPUs
- Should split a bit for GPUs too!
  - When starvation comes
  - How to evaluate starvation?

- Looking at « upward time » / « bottom level »
  - Duration of the critical path from considered task
  - With affine `for` loops etc., seems computable with polyhedral model
  - If higher than the remaining work, starvation is probable

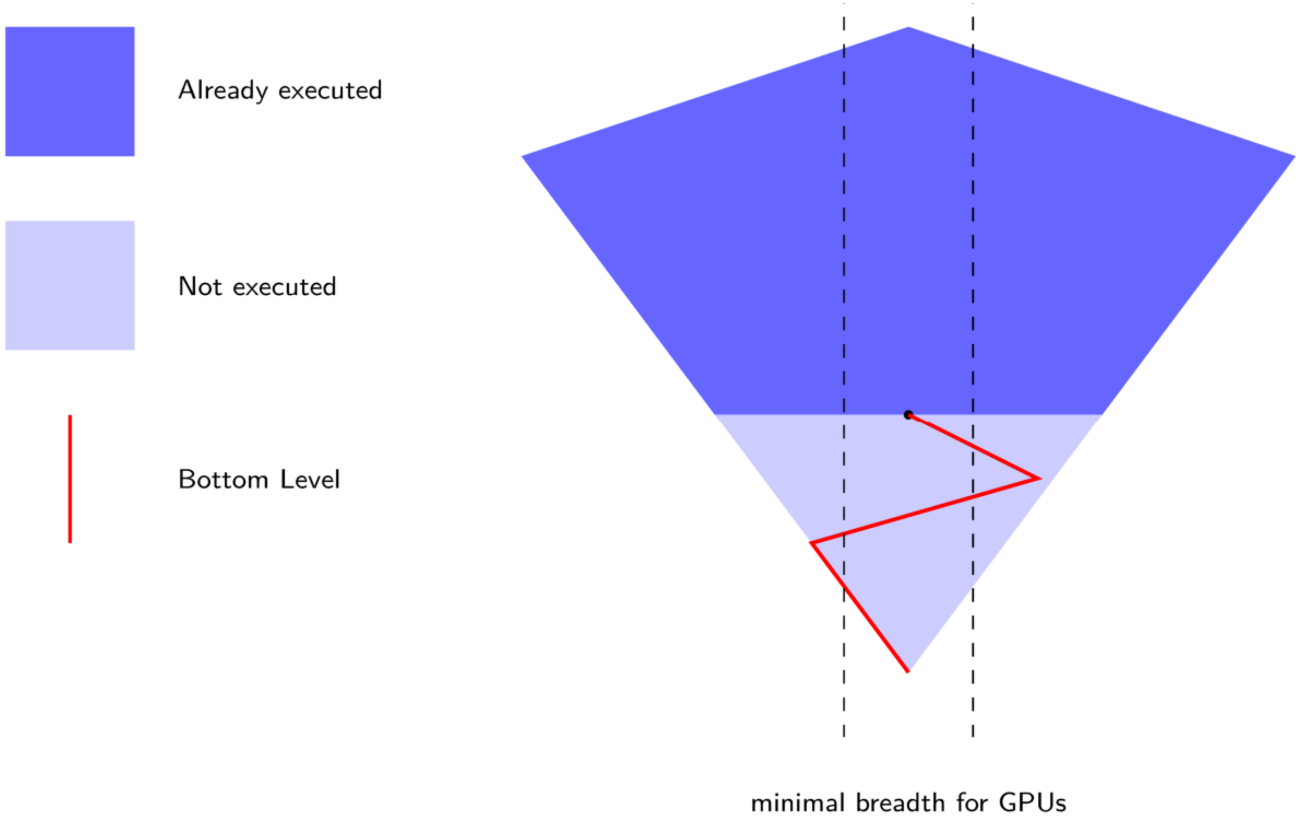
Current results : global view?



Current results : global view?



Current results : global view?



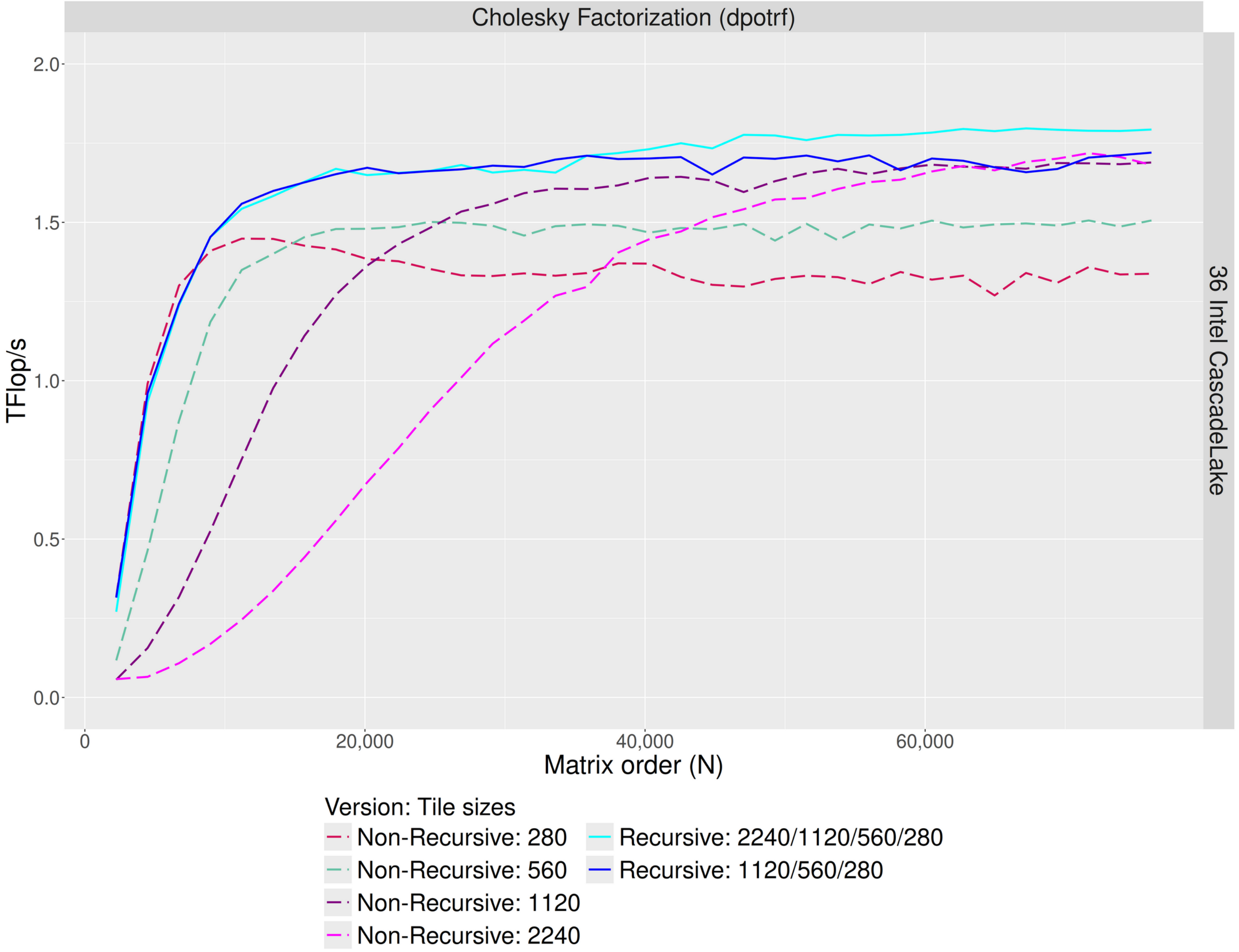
Current results : global view?

Estimating available parallelism :

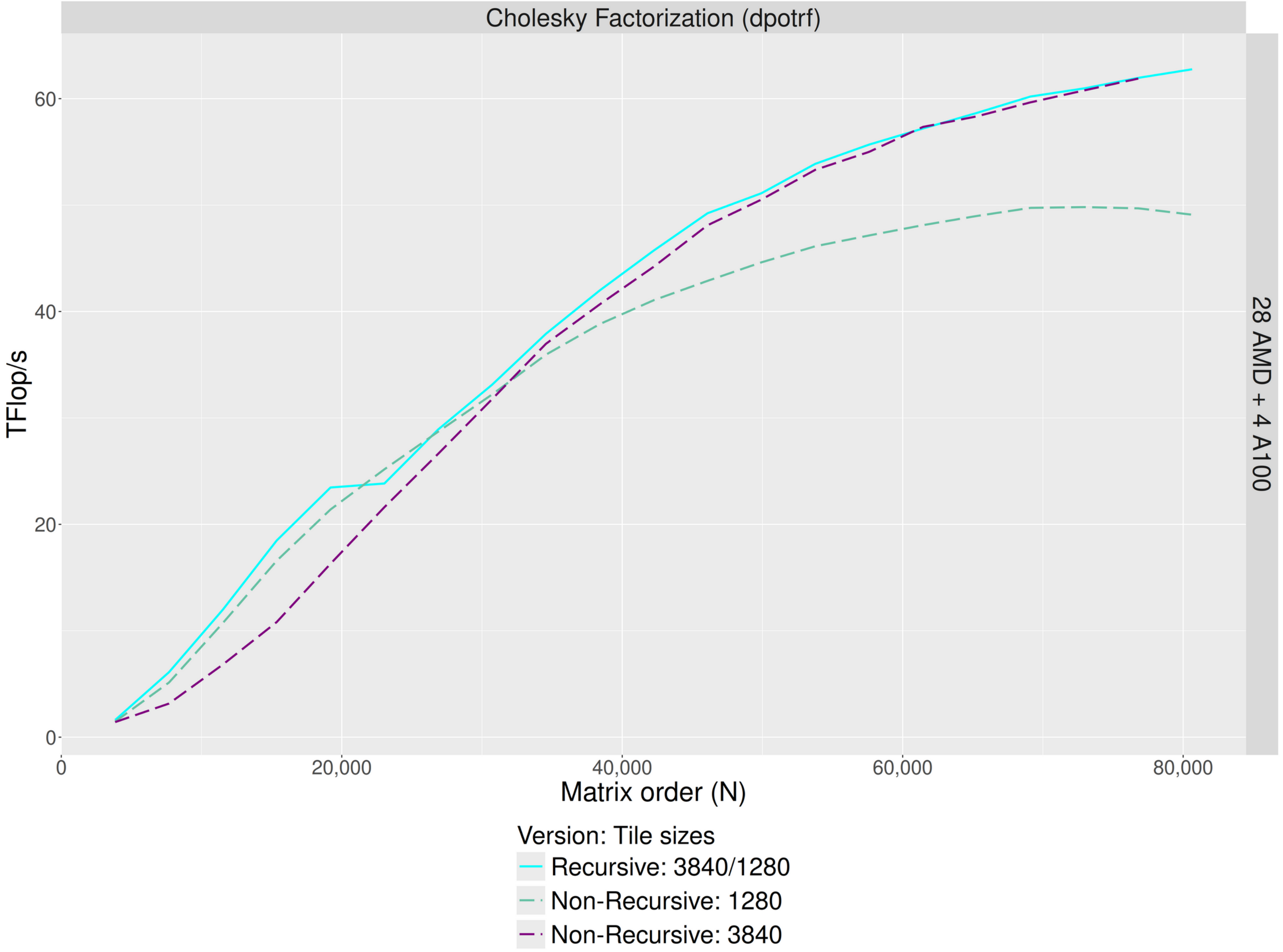
- work_time_left / critical_path_time(T)
- If less than number of GPUs, we should split T
  - To reduce critical path time
  - And create parallelism

For GPUs, can add data transfer time of T in critical path time

Current results : global view? CPUs



Current results : global view? GPUs



Conclusion

Recursive tasks

- A promising tool to steer granularity

Scheduling approaches

- Linear Program to get global view
- Greedy with local view, using Splitting profiles
- Greedy with global view using bottom level

How much global view are we willing to spend time looking at?

Perspectives

- Try with more irregular graphs
- Compute bottom levels with polyhedral model
- Scaling over MPI
  - Leverage recursivity for automatic pruning
  - Promising results with partial-pivoting LU
  - Hierarchical Load-balancing?