RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030 PROGRAMME DE RECHERCHE NUMÉRIQUE POUR L'EXASCALE

Recursive tasks

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

Summary

1. Recursive tasks
2. Granularity steering
3. Results

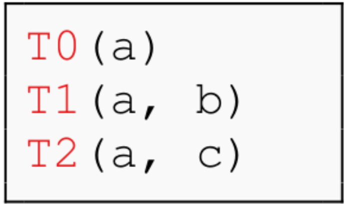
1. Recursive tasks

STF : Sequential Task Flow

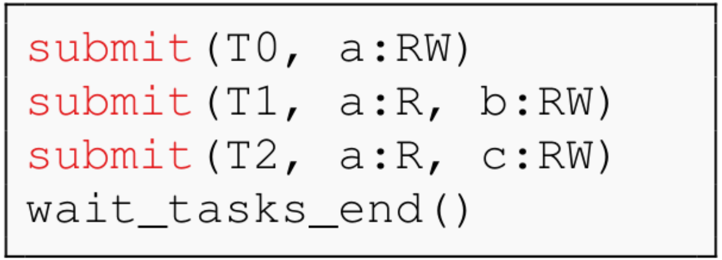
Natural way to express tasks

Dependencies

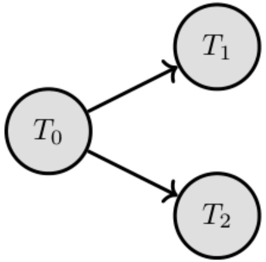
- Automatically inferred
- Order of submission



```
T0(a)
T1(a, b)
T2(a, c)
```

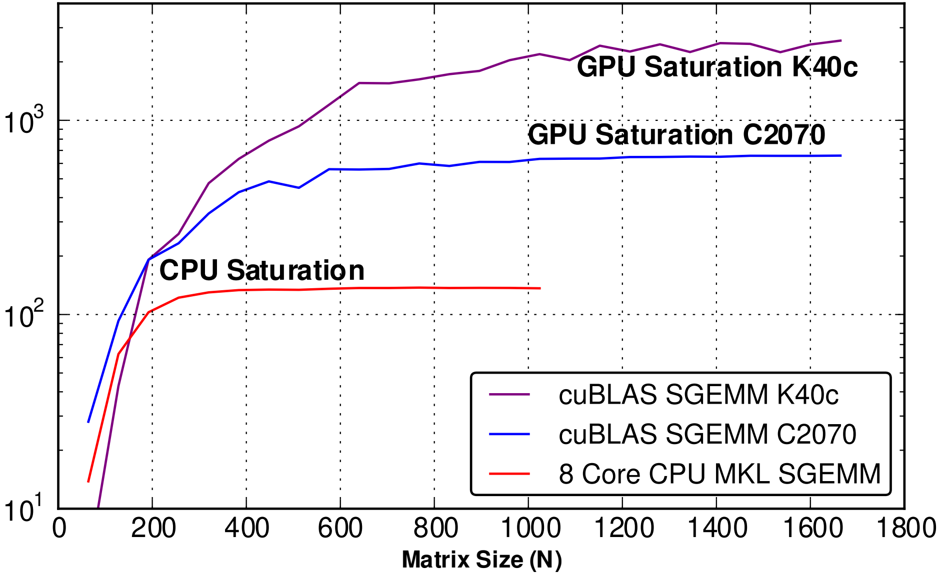


```
submit(T0, a:RW)
submit(T1, a:R, b:RW)
submit(T2, a:R, c:RW)
wait_tasks_end()
```



How big should a task be?

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



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

How big should a task be?

GPUs

- Efficient only with large tile sizes

CPUs

- Need many tasks

→ **Hybrid** task sizes

More generally, **recursive** task graphs

- Seen at CEA, in OmpSs, PaRSEC, StarPU

STF : Sequential Task Flow, recursive version

```
float V[256] ;
starpu_handle vect;
starpu_handle svec[PARTS] ;

int main(void)
{
    vector_data_register(&vec, V, 256, sizeof(*v)) ;
    data_partition_plan(vec, PARTS, svec) ;
    submit_tasks(svec) ;
    ...
}
```

STF : Sequential Task Flow, recursive version

```
void submit_tasks(starpu_data_handle_t *handles)
{
    for (int i = 0 ; i<PARTS ; i++)
        starpu_task_insert(&vector_scal, STARPU_RW, svec[i], 0) ;
}
```

STF : Sequential Task Flow, recursive version

```
float V[256] ;
starpu_handle vect;
starpu_handle svec[PARTS];

int main(void)
{
    vector_data_register(&vec, V, 256, sizeof(*v));
    data_partition_plan(vec, PARTS, svec);
    for (int i = 0 ; i < PARTS ; i++)
        data_partition_plan(svec[i], PARTS, &ssvec[i*PARTS]);
    submit_tasks(parent_arg, 1);
    …
}
```

STF : Sequential Task Flow, recursive version

```
int is_rec(struct rec_args *arg) {
    return arg->subparts[0] != NULL;
}
void submit_tasks(struct rec_arg *rec_args, int n) {
    for (int i=0 ; i<n ; i++)
        starpu_task_insert(&vector_scal,
            FUNC_REC, &is_rec, REC_ARG, rec_args[i],
            GEN_DAG, &gen_dag, GEN_ARG, rec_args[i],
            STARPU_RW, rec_args[i]->h, 0);
}
void gen_dag(struct rec_arg *args) {
    submit_tasks(arg->subparts,  PARTS);
}
```

STF : Sequential Task Flow, recursive version

Can leverage tiled algorithm expression

- In Chameleon, just structure sugar around existing tiled algorithms
- Immediately get various recursive task graphs
    - potrf, getrf, poinv, posv, potri…

And let runtime decide how deep to recurse

- Larger tasks for GPUs
- Smaller tasks for CPUs

Ideally, let runtime decide among a scale of granularities

- e.g. 7680 / 3840 / 1920 / 960 / 480

Can’t we rather use parallel tasks?

- Large tasks for GPUs
- Parallel tasks on CPUs

Parallel tasks are not perfect

- e.g. idle time at beginning and end of POTRF
- better expose the inner lack of parallelism
- i.e. the subtaskgraph
  - To overlap the lack of parallelism

2. Granularity steering

Illustrative example

2 GPUs way faster than 1 CPU core

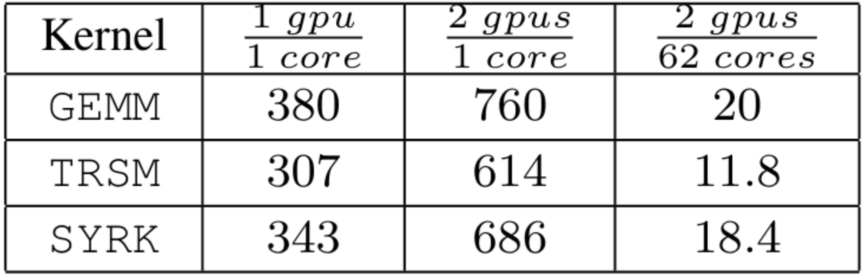
- Need 760 GEMMs to have some to give to CPU

2 GPUs not that faster than 62 CPU cores

- With 21 GEMMs, can afford splitting one for CPU

But ratios depend on ready tasks types

- Better split TRSM tasks
- e.g. split 0% GEMM, 0% SYRK, 70% TRSM



Kernel	$\frac{1\ gpu}{1\ core}$	$\frac{2\ gpus}{1\ core}$	$\frac{2\ gpus}{62\ cores}$
GEMM	380	760	20
TRSM	307	614	11.8
SYRK	343	686	18.4

Finding splitting ratios?

Depends on situation

- Lot of parallelism available
  - No need to split
  - Can run large tasks on CPUs
  - Leverage largest-tile choices
- Lacking parallelism
  - Produce parallelism while keeping an eye on efficiency
- Availability of different task types
  - Better split tasks that split efficiently

Linear programming

Minimize

$$exT$$

Subject to

Task number splitting.

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

No last−level splitting.

$$Ns^t_{\mathcal{L}} = 0 \qquad \forall\, t\in\mathcal{T} \qquad (2)$$

Compl. time when executing tasks.

$$\sum_{\substack{t\in\mathcal{T}\\ 0\leq l\leq\mathcal{L}}} Ne^t_{l,u} \cdot Ex^u_{t,l} - Idle_u \cdot R^u \cdot exT \leq 0 \qquad u\in\mathcal{R} \qquad (3)$$

Minimal number of tasks on PU type.

$$\sum_{\substack{t\in\mathcal{T}\\ 0\leq l\leq\mathcal{L}}} Ne^t_{l,u} - R^u \cdot MinN_u \leq 0 \qquad u\in\mathcal{R} \qquad (4)$$

Linear programming

- Takes something like 0.1ms-2ms to solve (GLPK) for the tested matrices
- Can afford solving it every 50 tasks for instance

Result:

- Splitting ratio for each type of task
  - According to the **current** state of ready tasks

Splitter

- Strive to reach the ratios, and progressively
- Split if
  - ratio not met yet
  - and not enough tasks for CPUs

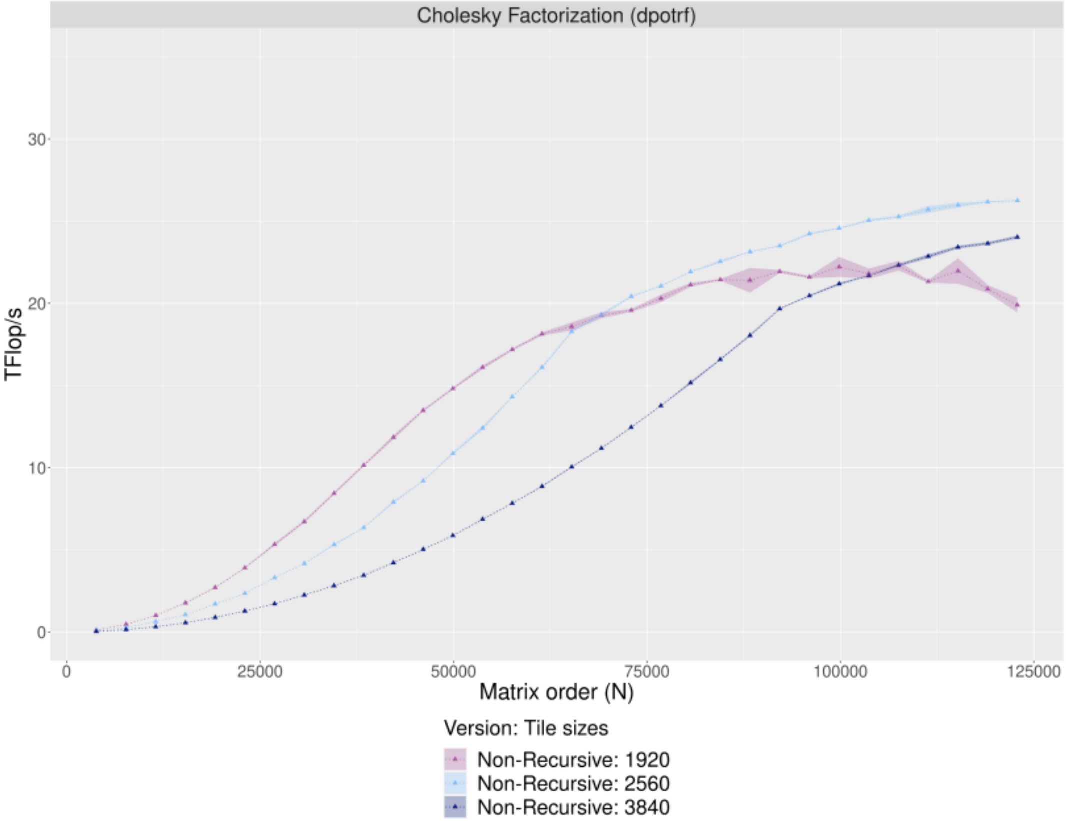
3. Results

Experiments

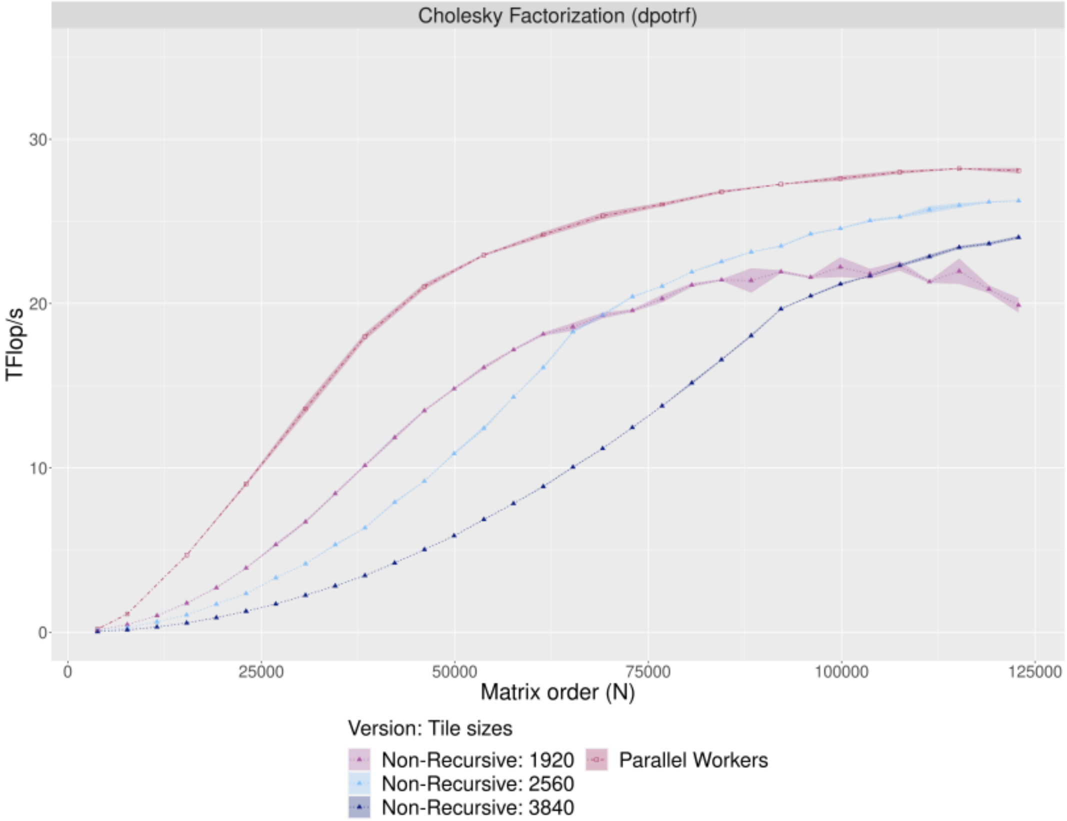
Cholesky, LU, and other po*

- 2 A100 GPUs
- 2 AMD Zen3 EPYC 7513 2.6 GHz 32 cores → 64 cores
- DMDAR scheduler

Cholesky

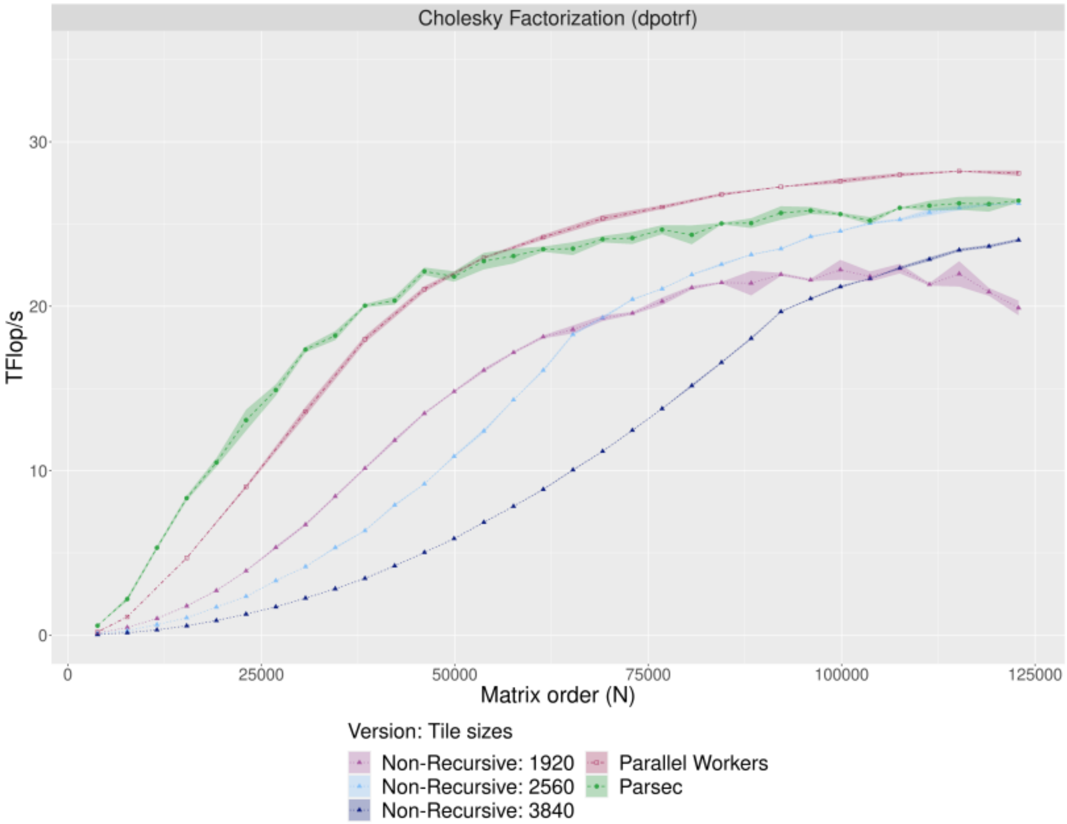

Cholesky Factorization (dpotrf)
TFlop/s
0
10
20
30
0
25000
50000
75000
100000
125000
Matrix order (N)
Version: Tile sizes
Non-Recursive: 1920
Non-Recursive: 2560
Non-Recursive: 3840

Cholesky

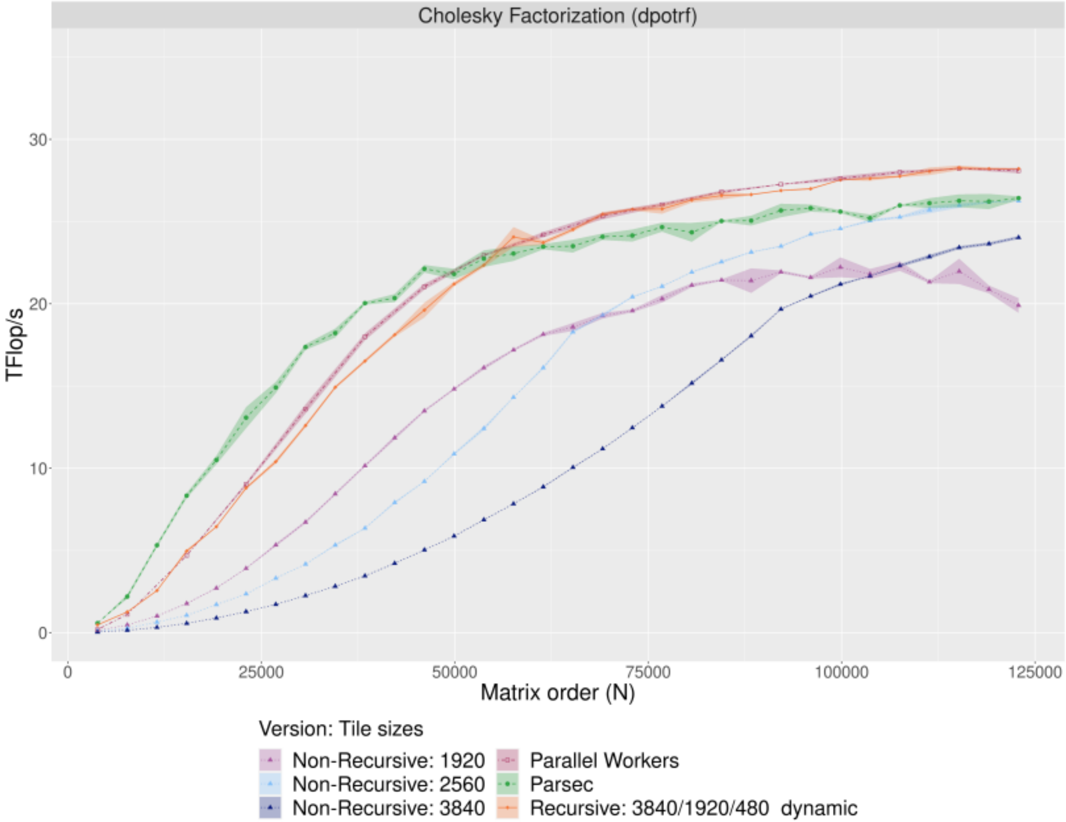


Cholesky Factorization (dpotrf)
TFlop/s
0
10
20
30
0
25000
50000
75000
100000
125000
Matrix order (N)
Version: Tile sizes
Non-Recursive: 1920
Non-Recursive: 2560
Non-Recursive: 3840
Parallel Workers

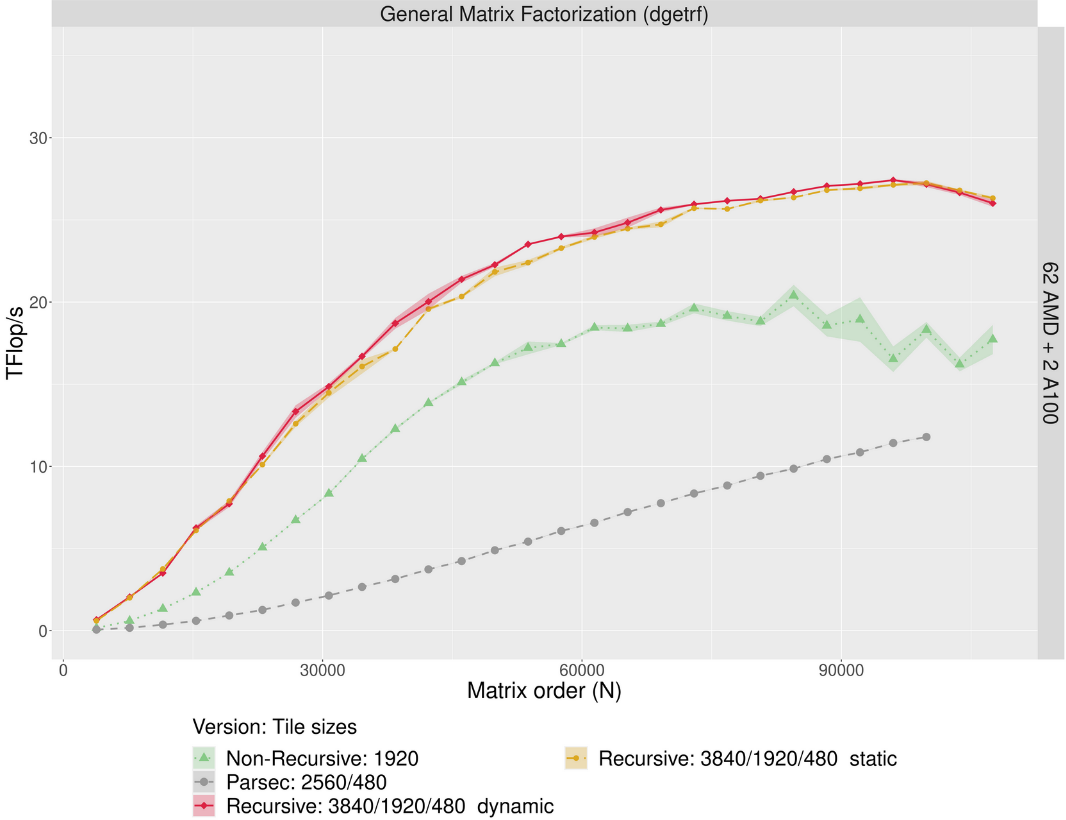
Cholesky

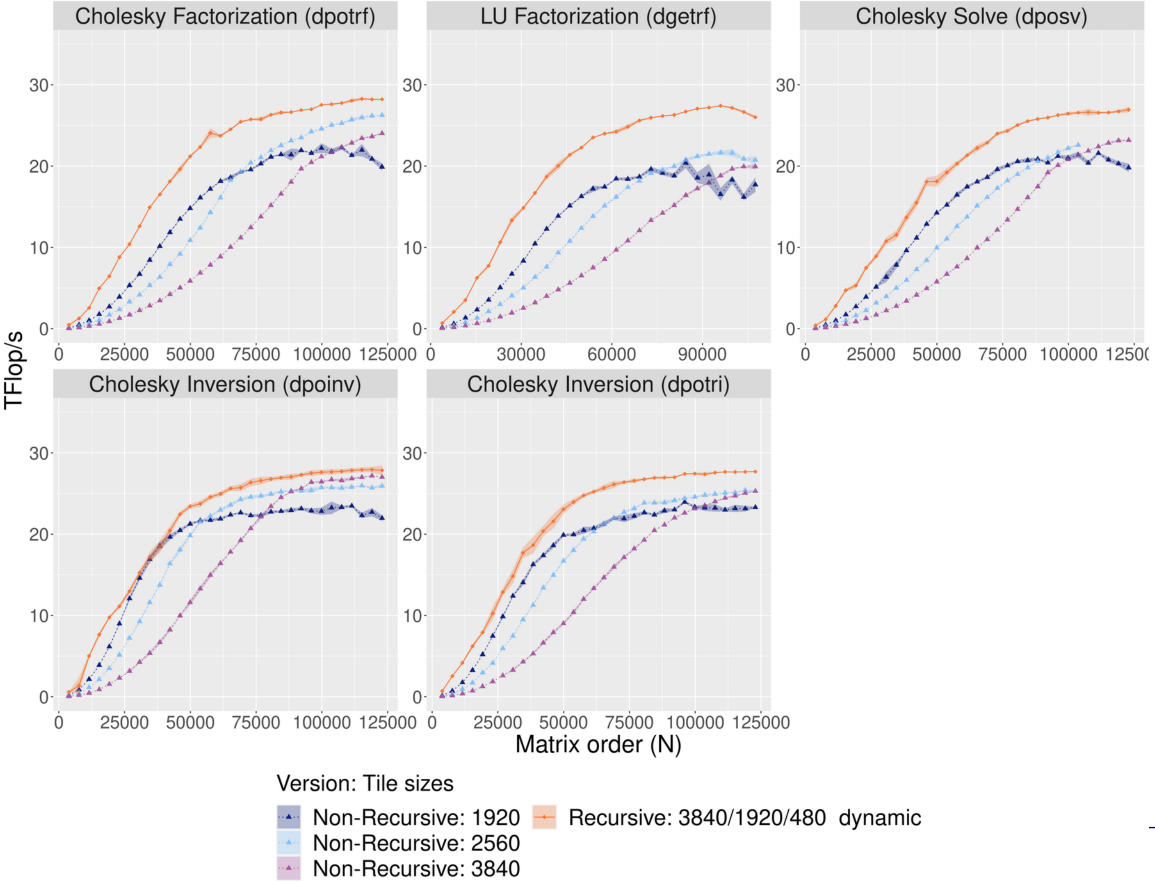

Cholesky Factorization (dpotrf)
TFlop/s
0
10
20
30
Matrix order (N)
0
25000
50000
75000
100000
125000
Version: Tile sizes
Non-Recursive: 1920
Non-Recursive: 2560
Non-Recursive: 3840
Parallel Workers
Parsec

Cholesky


Cholesky Factorization (dpotrf)
TFlop/s
0
10
20
30
Matrix order (N)
0
25000
50000
75000
100000
125000
Version: Tile sizes
Non-Recursive: 1920
Non-Recursive: 2560
Non-Recursive: 3840
Parallel Workers
Parsec
Recursive: 3840/1920/480 dynamic

LU


General Matrix Factorization (dgetrf)
62 AMD + 2 A100
TFlop/s
0
10
20
30
0
30000
60000
90000
Matrix order (N)
Version: Tile sizes
Non-Recursive: 1920
Parsec: 2560/480
Recursive: 3840/1920/480 dynamic
Recursive: 3840/1920/480 static


Cholesky Factorization (dpotrf)
LU Factorization (dgetrf)
Cholesky Solve (dposv)
Cholesky Inversion (dpoinv)
Cholesky Inversion (dpotri)
TFlop/s
0
10
20
30
0
25000
50000
75000
100000
125000
30000
60000
90000
Matrix order (N)
Version: Tile sizes
Non-Recursive: 1920
Non-Recursive: 2560
Non-Recursive: 3840
Recursive: 3840/1920/480 dynamic

Conclusion

Recursive task graphs

- Flexible way to express parallelism
- Let runtime decide granularity

Dynamic granularity decision

- Adapts to runtime situation
    - No manual tuning
- On par with state-of-the-art performance

Future work

Refine splitting decision

- Take care of critical path in the task graph

Leverage compilation

- Generate recursive expression automatically?

Going distributed

- Allow automatic pruning?

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030
PROGRAMME
DE RECHERCHE
NUMÉRIQUE
POUR L'EXASCALE

Retrouvez toutes nos actualités

in **NumPEx**