RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Dynamic Task Graph Adaptation with Recursive Tasks

Exa-Soft

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

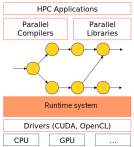
September 2025

Introduction

Introduction - Context

Task-based Programming

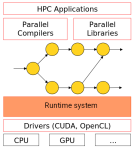
- Motivations:
  - Portable frameworks.
  - Exploit complex architectures.
- Applications: Directed Acyclic Graph (DAG).
- Runtime systems: scheduling, data management, communications, ...



Introduction - Context

Task-based Programming

- Motivations:
  - Portable frameworks.
  - Exploit complex architectures.
- Applications: Directed Acyclic Graph (DAG).
- Runtime systems: scheduling, data management, communications, ...

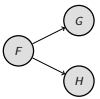


STF: Sequential Task Flow

- Dependencies:
  - Automatically inferred.
  - Order of submission.

```
F(a)
G(a, b)
H(a, c)
```

```
submit(F, a:RW)
submit(G, a:R, b:RW)
submit(H, a:R, c:RW)
wait_tasks_completion()
```



Introduction - Limitations of the STF model

Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

Introduction - Limitations of the STF model

Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

$\Rightarrow$ How to create more dynamic task-graphs ?

Introduction - Limitations of the STF model

Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

$\Rightarrow$ How to create more dynamic task-graphs ? $\Rightarrow$ Recursive tasks graphs !

Introduction - Limitations of the STF model

Submission

- Overhead: large number of non-ready tasks.
- Bottleneck: sequential insertion.
- Adaptability ? static task graphs.

⇒ How to create more dynamic task-graphs ? ⇒ Recursive tasks graphs !

Granularity

- GPUs versus CPUs.
- Lack of parallelism versus Steady State.

⇒ Steering granularity dynamically ?

Overview

- StarPU's Recursive Tasks.
- Introduction of the Splitter component.
- The heterogeneous case:
  - A first policy that relies on Linear Programming.
  - Improving this policy with a greedy algorithm.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

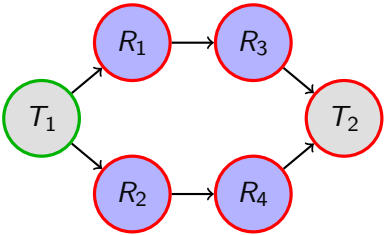
PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

cea cnrs Inria

Recursive Tasks

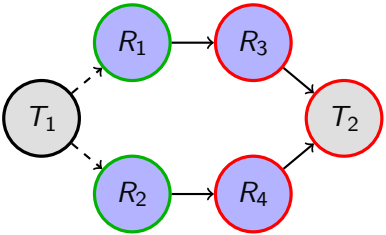
Recursive Tasks in StarPU

Recursive Tasks in StarPU



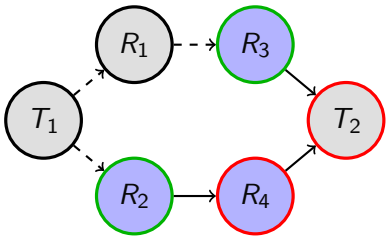
Recursive Tasks in StarPU

- Recursive task execution:



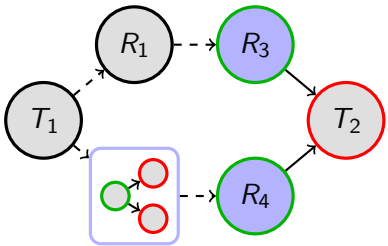
Recursive Tasks in StarPU

- Recursive task execution:
  - Remain regular task.



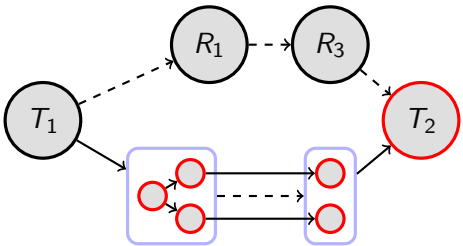
Recursive Tasks in StarPU

- Recursive task execution:
  - Remain regular task.
  - Insert a subgraph: **split**.



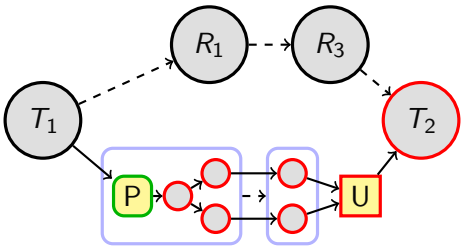
Recursive Tasks in StarPU

- Recursive task execution:
  - Remain regular task.
  - Insert a subgraph: **split**.



Recursive Tasks in StarPU

- Recursive task execution:
  - Remain regular task.
  - Insert a subgraph: **split**.



RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

FRANCE 2030

PROGRAMME DE RECHERCHE
NUMÉRIQUE POUR L'EXASCALE

cea cnrs Inria

Dynamic task graph adaptation

Dynamic task graph adaptation : splitting tasks

When do we choose to split task?

Which task should we split?

Dynamic task graph adaptation : splitting tasks

When do we choose to split task?

Submission, execution, ...

Which task should we split?

Dynamic task graph adaptation : splitting tasks

When do we choose to split task?

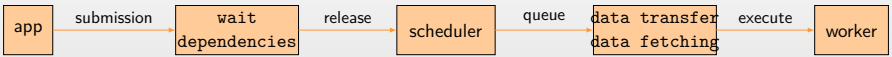
Submission, execution, ...

Which task should we split?

Efficiency, Completion Time, Current Parallelism

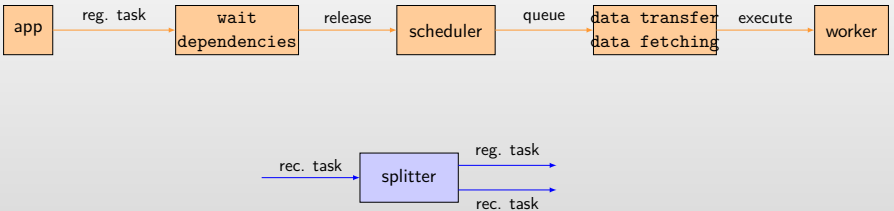
When do we choose to split tasks

Task life path



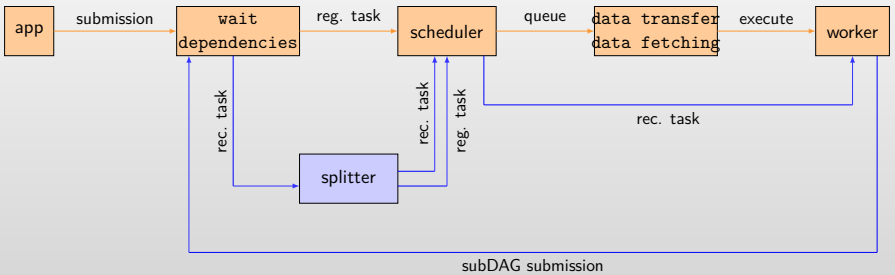
When do we choose to split tasks

Adding the splitter



When do we choose to split tasks

Position of the splitter - trade-off



Heterogeneous

Which task do we split - Heterogeneous case - Problematic

General example:

- Heterogeneous platform with 2 GPU Nvidia A100 and 2 AMD Zen3 CPU, each with 32 cores.

Which task do we split - Heterogeneous case - Problematic

General example:

- Heterogeneous platform with 2 GPU Nvidia A100 and 2 AMD Zen3 CPU, each with 32 cores.
- Performing a tiled Cholesky factorization.

Kernel	$\frac{1\ gpu}{1\ core}$	
GEMM	380	

Which task do we split - Heterogeneous case - Problematic

General example:

- Heterogeneous platform with 2 GPU Nvidia A100 and 2 AMD Zen3 CPU, each with 32 cores.
- Performing a tiled Cholesky factorization.

Kernel	$\frac{1\ gpu}{1\ core}$	$\frac{1\ gpu}{StarPU\ 64\ cores}$ (accel.)
GEMM	380	9.4 (40.4)

Which task do we split - Heterogeneous case - Problematic

General example:

- Heterogeneous platform with 2 GPU Nvidia A100 and 2 AMD Zen3 CPU, each with 32 cores.
- Performing a tiled Cholesky factorization.

Kernel	$\frac{1\ gpu}{1\ core}$	$\frac{1\ gpu}{StarPU\ 64\ cores}$ (accel.)
GEMM	380	9.4 (40.4)
TRSM	307	6.5 (45.8)
SYRK	343	11.7 (29.3)

Which task do we split - Heterogeneous case - Problematic

General example:

- Heterogeneous platform with 2 GPU Nvidia A100 and 2 AMD Zen3 CPU, each with 32 cores.
- Performing a tiled Cholesky factorization.

Kernel	$\frac{1\ gpu}{1\ core}$	$\frac{1\ gpu}{StarPU\ 64\ cores}$ (accel.)
GEMM	380	9.4 (40.4)
TRSM	307	6.5 (45.8)
SYRK	343	11.7 (29.3)

- Q: Steady state $\Rightarrow$ Which tasks, and how many should be divided ?

Which task do we split - Heterogeneous case

- A task becomes ready $\Rightarrow$ Recorded by the Splitter.

Which task do we split - Heterogeneous case

- A task becomes ready $\Rightarrow$ Recorded by the Splitter.
- Each 50-task interval $\Rightarrow$ A Linear Program is called.

Which task do we split - Heterogeneous case

- A task becomes ready $\Rightarrow$ Recorded by the Splitter.
- Each 50-task interval $\Rightarrow$ A Linear Program is called.
- Linear Program's aim: minimizing execution time by balancing load between the PUs.

Which task do we split - Heterogeneous case

- A task becomes ready $\Rightarrow$ Recorded by the Splitter.
- Each 50-task interval $\Rightarrow$ A Linear Program is called.
- Linear Program's aim: minimizing execution time by balancing load between the PUs.
- Task Splitting for PUs is required by ensuring minimal parallelism for each type of PU.

Which task do we split - Heterogeneous case

- A task becomes ready ⇒ Recorded by the Splitter.
- Each 50-task interval ⇒ A Linear Program is called.
- Linear Program's aim: minimizing execution time by balancing load between the PUs.
- Task Splitting for PUs is required by ensuring minimal parallelism for each type of PU.
- The LP provides ratio ⇒ used by the Splitter.

Which task do we split - Heterogeneous case - Linear Program

Parameters

$\mathcal{T}$, $N^{tot}_{t,l}$	$t \in \mathcal{T}$	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^t_l	$t \in \mathcal{T}, l < \mathcal{L}$	Number of split task of type t and level l.
$Ne^t_{l,u}$	$t \in \mathcal{T}, l \leq \mathcal{L}, u \in \mathcal{R}$	Number of task of type t and level l executed on PU u

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.

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

Benchmarks - Heterogeneous

The tests were run on PlaFRIM's `sirocco` nodes:

- 2x 32-core AMD Zen3 EPYC 7513 @ 2.6 GHz
- 2x NVIDIA A100 (40GB)
- 512 GB (8 GB/core) (@3200 MHz)
- Scheduler : Deque Model Data-Aware Ready (DMDAR)

Benchmarks - Heterogeneous

The tests were run on PlaFRIM's `sirocco` nodes:

- 2x 32-core AMD Zen3 EPYC 7513 @ 2.6 GHz
- 2x NVIDIA A100 (40GB)
- 512 GB (8 GB/core) (@3200 MHz)
- Scheduler : Deque Model Data-Aware Ready (DMDAR)

Tile sizes choosen :

- 3840 : "big" : the most efficient.
- 480 : "small": parallelism, for CPU cores.
- 1920, 2560 : "mid": trade-off.

Benchmarks - Heterogeneous - Cholesky

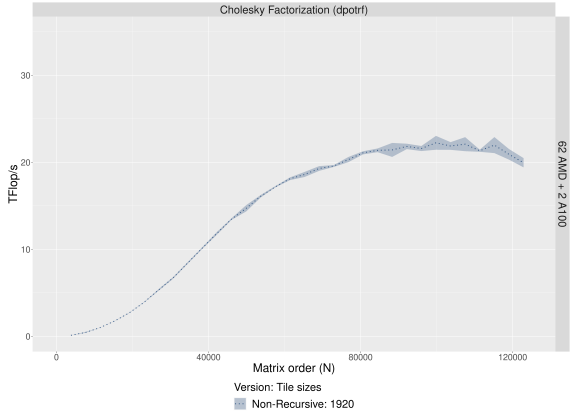


Figure 1: Performance comparison between different versions for Cholesky Factorization.

Benchmarks - Heterogeneous - Cholesky

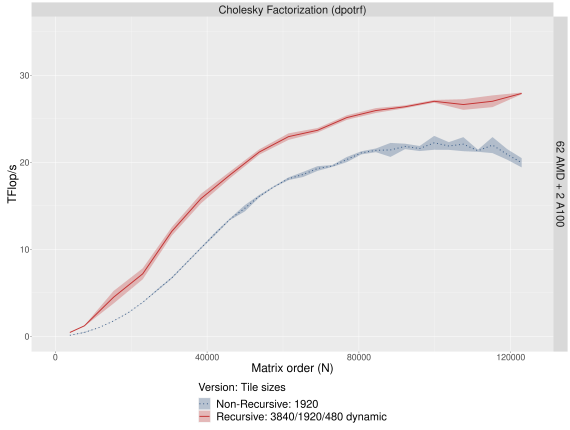


Figure 1: Performance comparison between different versions for Cholesky Factorization.

Benchmarks - Heterogeneous - Cholesky

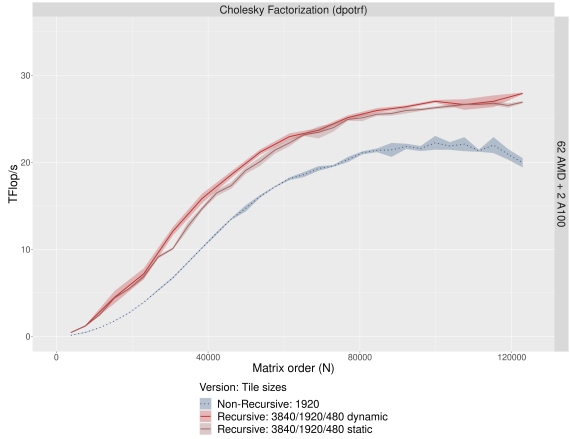


Figure 1: Performance comparison between different versions for Cholesky Factorization.

Benchmarks - Heterogeneous - Cholesky

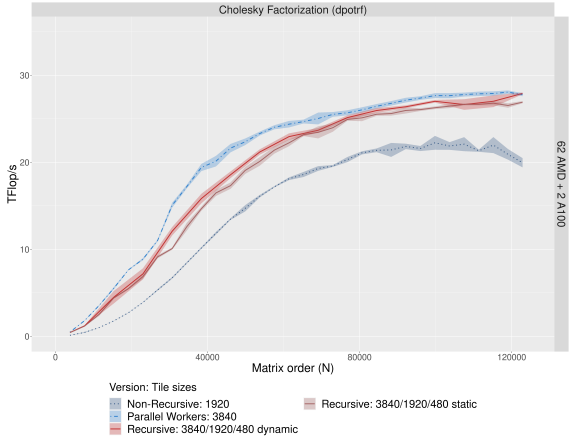


Figure 1: Performance comparison between different versions for Cholesky Factorization.

Benchmarks - Heterogeneous - Cholesky

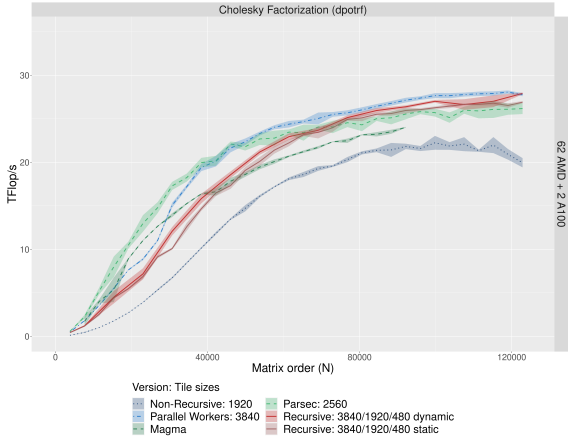


Figure 1: Performance comparison between different versions for Cholesky Factorization.

Benchmarks - Heterogeneous - without piv

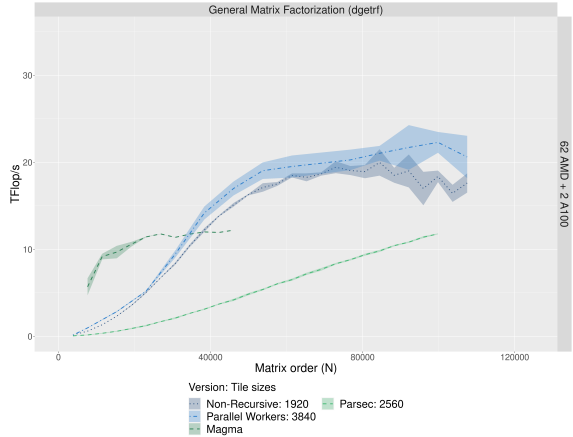


Figure 2: Performance comparison between different versions for LU Factorization without piv.

Benchmarks - Heterogeneous - without piv

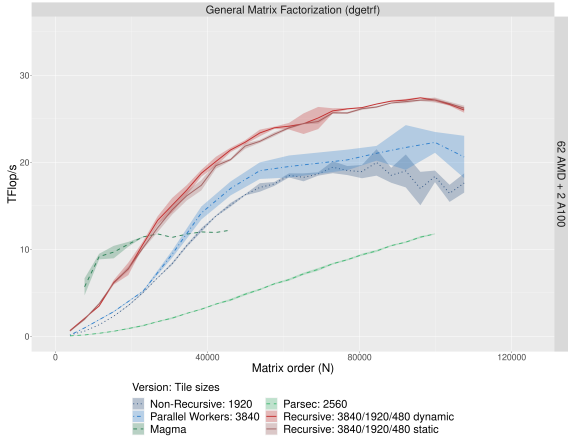


Figure 2: Performance comparison between different versions for LU Factorization without piv.

Benchmarks - Heterogeneous - All

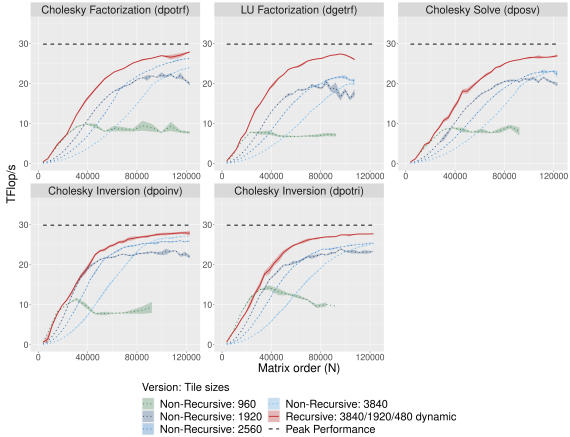


Figure 3: Performance comparison between different kernels and versions.

Benchmarks - Heterogeneous Cholesky trace

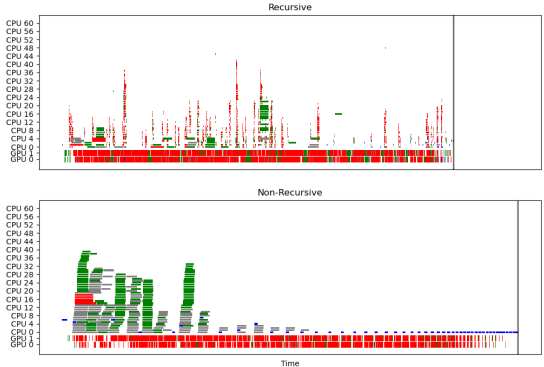


Figure 4: Comparison of traces between Recursive (3840/1920/480) and Non-Recursive (1920) versions for a Cholesky.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.
  - At least same performance as non-recursive versions.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.
  - At least same performance as non-recursive versions.
- Inconvenients:
  - Decisions taken by looking at past. *Cf. call interval*.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.
  - At least same performance as non-recursive versions.
- Inconvenients:
  - Decisions taken by looking at past. *Cf. call interval*.
  - Opaque decision.

Which task do we split - Heterogeneous case

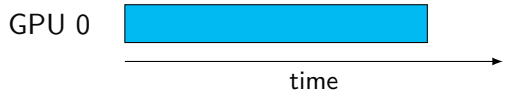
- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.
  - At least same performance as non-recursive versions.
- Inconvenients:
  - Decisions taken by looking at past. *Cf. call interval*.
  - Opaque decision.
  - CPU usage can be increased.

Which task do we split - Heterogeneous case

- Summary:
  - Automatic + Generic $\Rightarrow$ Granularity dynamically steered.
  - Dense linear algebra problems:
    - Accross non-recursive Chameleon: Improvements from 5 to 30 %.
    - Accross state-of-the-arts versions: on-par results in a portable way.
- Advantages:
  - Generic.
  - Automatic.
  - Requirements: Recursive implementations + Performance models.
  - At least same performance as non-recursive versions.
- Inconvenients:
  - Decisions taken by looking at past. *Cf. call interval*.
  - Opaque decision.
  - CPU usage can be increased.
  - Turning a global view into a decision with local constraints.

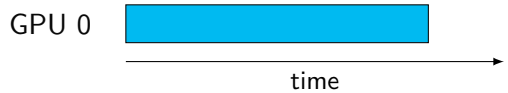
A splitting example

A splitting example



A splitting example

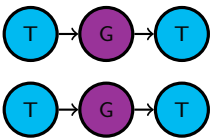
Baseline scheduling:



A splitting example

Baseline scheduling:

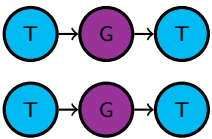
Potential split:



A splitting example

Baseline scheduling:

Potential split:

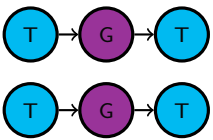


Potential scheduling:

A splitting example

Baseline scheduling:

Potential split:

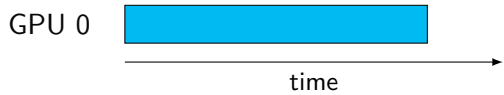


Potential scheduling:

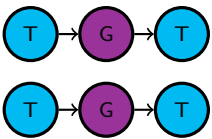
Not interesting: longer than baseline.

A splitting example

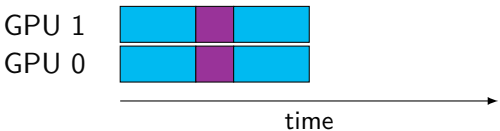
Baseline scheduling:



Potential split:



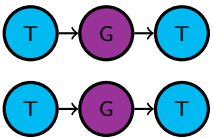
Potential scheduling:



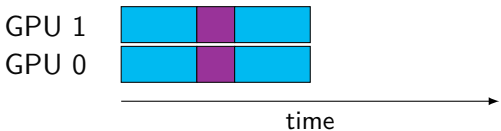
A splitting example

Baseline scheduling:

Potential split:



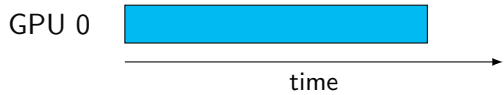
Potential scheduling:



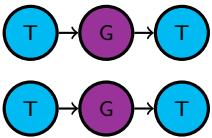
gpu-time increased, end-time decreased.

A splitting example

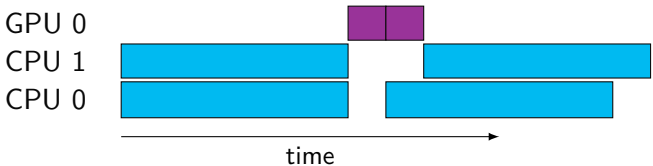
Baseline scheduling:



Potential split:

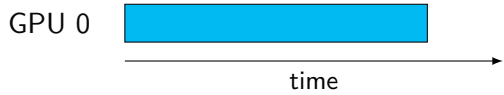


Potential scheduling:

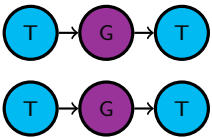


A splitting example

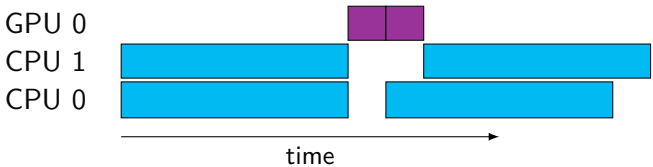
Baseline scheduling:



Potential split:



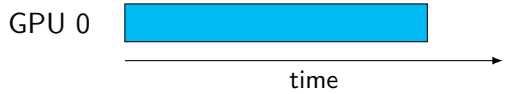
Potential scheduling:



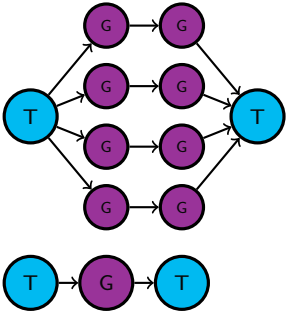
End-time increased, GPU time decreased.

A splitting example

Baseline scheduling:

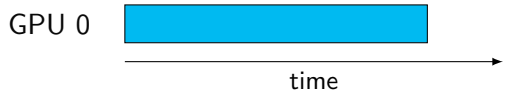


Potential split:

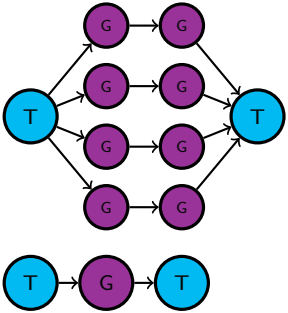


A splitting example

Baseline scheduling:



Potential split:

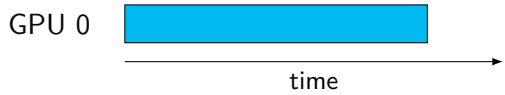


Idea: for each type of task, create potential interesting splittings and schedulings:

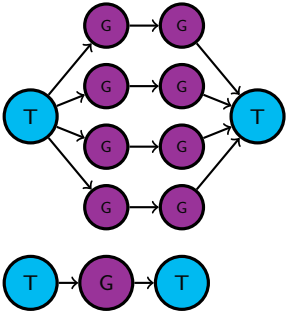
- Decrease GPU time.
- Occupy CPU cores.

A splitting example

Baseline scheduling:



Potential split:



Idea: for each type of task, create potential interesting splittings and schedulings:

- Decrease GPU time.
- Occupy CPU cores.

Record, for each scheduling:

- Sequential GPU time.
- Global finishing time.
- Mean of CPU cores used.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.
  - Else, take the less accelerated task,

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.
  - Else, take the **less accelerated task**,

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.
  - Else, take the **less accelerated task**, and search for a recursive scheduling that

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.
  - Else, take the **less accelerated task**, and search for a recursive scheduling that
    - Can finish before the GPUs.
    - Occupy less CPU cores than the available CPUs.

A new algorithm to improve previous limitations

- LP $\Rightarrow$ *global decision* **vs** greedy $\Rightarrow$ *incremental decision with a* ***global target***.
- Global target: when possible, splitting a task to occupy CPU cores and constraint the scheduling.
- At start, for each task generation of "all" recursive versions and "all" schedulings.
  - All **interesting** recursive versions and schedulings.
- When a task is about to be pushed, look at GPU end.
  - If no task can be split with a part executed on CPU cores, push it to GPUs.
  - Else, take the **less accelerated task**, and search for a recursive scheduling that
    - Can finish before the GPUs.
    - Occupy less CPU cores than the available CPUs.
- Warning: Splitting generates data transfers that should be taken into account.

Benchmarks - Heterogeneous - Cholesky

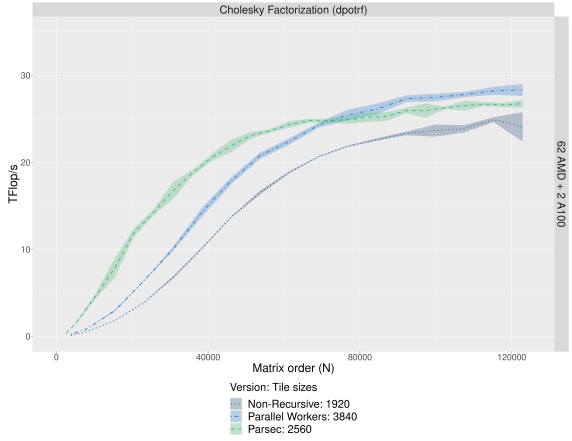


Figure 5: Performance comparison between different versions for Cholesky Factorization with Greedy Algorithm.

Benchmarks - Heterogeneous - Cholesky

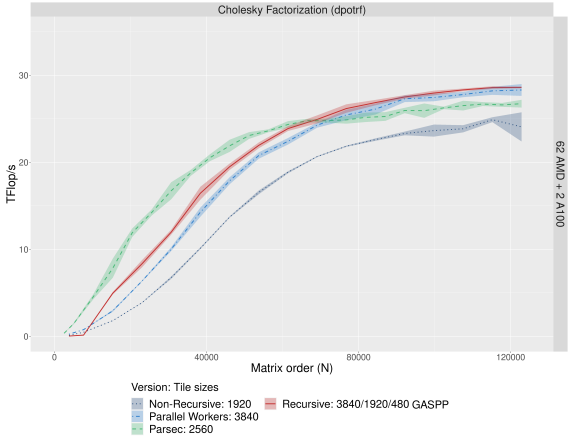


Figure 5: Performance comparison between different versions for Cholesky Factorization with Greedy Algorithm.

Benchmarks - Heterogeneous - Cholesky

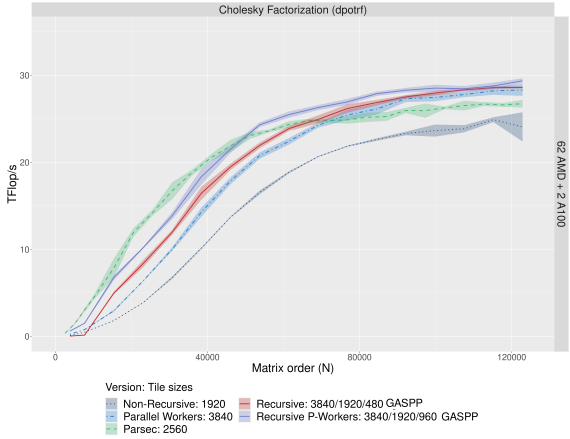


Figure 5: Performance comparison between different versions for Cholesky Factorization with Greedy Algorithm.

Benchmarks - Heterogeneous - LU without piv

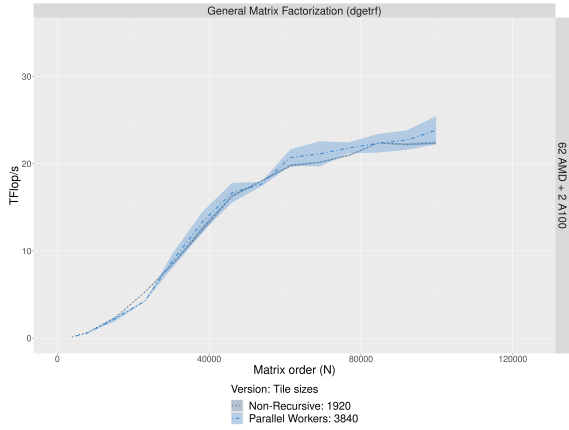


Figure 6: Performance comparison between different versions for LU Factorization without piv with Greedy Algorithm.

Benchmarks - Heterogeneous - LU without piv

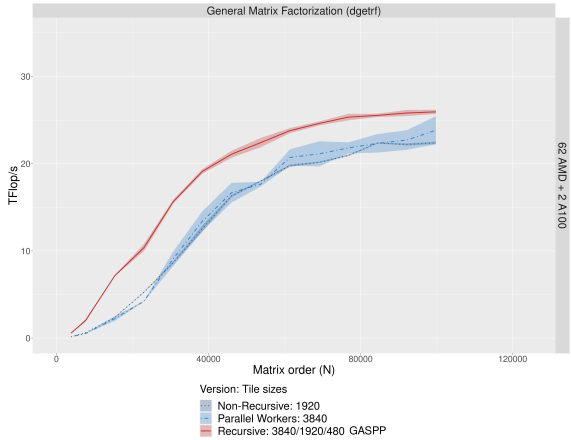


Figure 6: Performance comparison between different versions for LU Factorization without piv with Greedy Algorithm.

Benchmarks - Heterogeneous - LU without piv

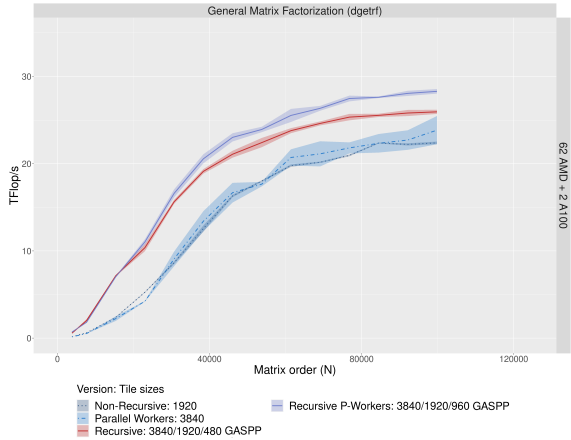


Figure 6: Performance comparison between different versions for LU Factorization without piv with Greedy Algorithm.

Benchmarks - Heterogeneous - All

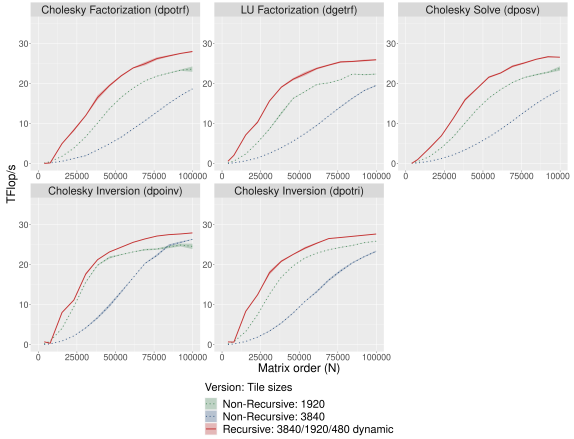


Figure 7: Performance comparison between different kernels and versions with Greedy Algorithm.

Benchmarks - Heterogeneous - POTRF on GPU

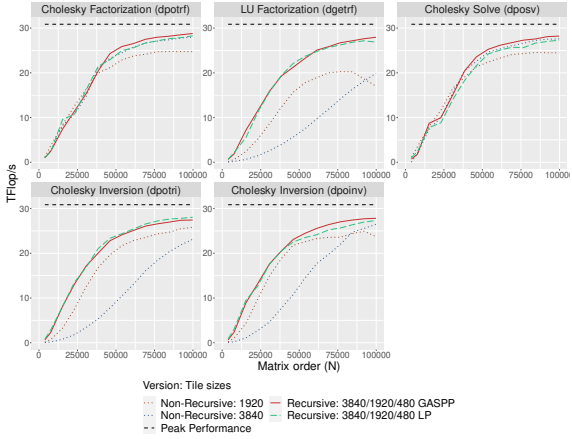


Figure 8: Performance comparison between different kernels and versions with Greedy Algorithm, having dpotrf on GPU.

Benchmarks - Heterogeneous - Cholesky - POTRF on GPU

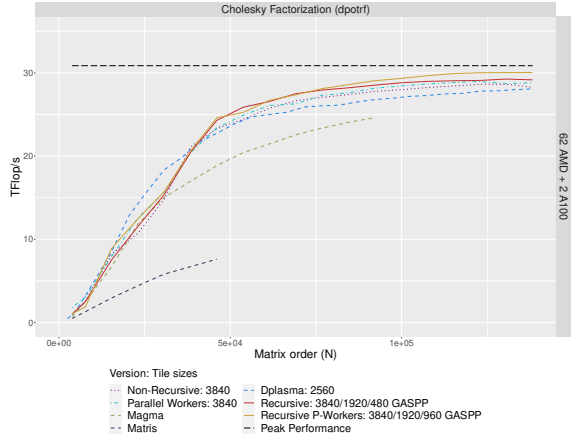


Figure 9: Performance comparison between different versions for Cholesky Factorization with Greedy Algorithm, having POTRF on GPU.

Conclusion

Conclusion

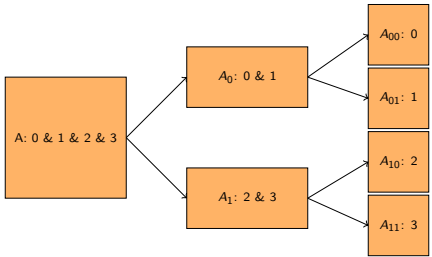
- Recursive tasks:
  - Insert subgraph at runtime.
  - More dynamic DAG.
- Splitting task dynamically brings different questions:
  - Which task sould we split.
  - When do we choose to split.

Current Work

- Distributed recursive tasks, with scheduling.
- Extend to more irregular applications.

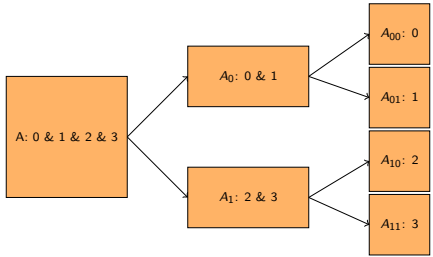
Distributed context

Shared data

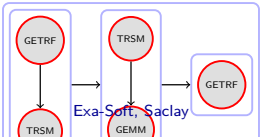


Distributed context

Shared data

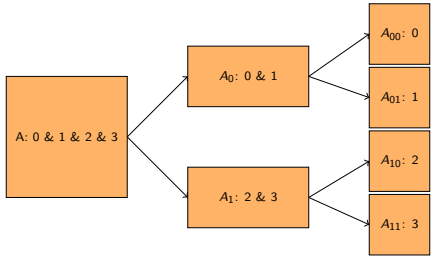


Auto-pruning

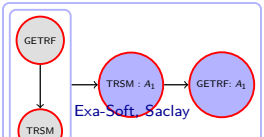


Distributed context

Shared data



Auto-pruning Node 0



Distributed context

Advantages

- Auto-pruning is important to decrease runtime pressure.
- Communications submitted Just-In-Time, reducing pressure.

Benchmarks - Homogeneous - LU nopiv

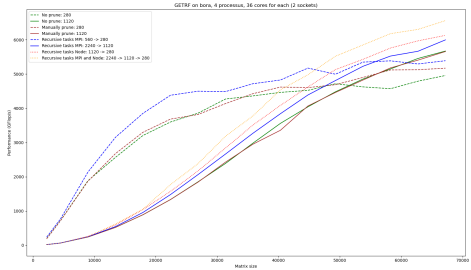


Figure 10: Performance comparison between different versions for LU nopiv.

- No prune: no pruning, all tasks inserted, StarPU remove not submit some.
- Manually prune: The app submits only the needed tasks.
- Recursive tasks MPI: All tasks are submitted, without pruning, but communications are inserted at splitting.
- Recursive tasks Node: The last quarter of the iterations are splitted on node to