DYNAMIC TASK GRAPH ADAPTATION WITH RECURSIVE TASKS

17^{th} Scheduling Workshop for large-scale systems @ Aussois

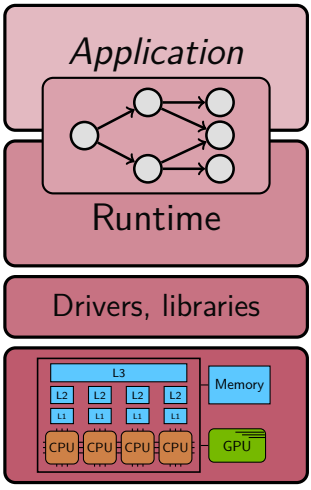
---

Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Raymond Namyst, Samuel Thibault, Pierre-André Wacrenier

INTRODUCTION

The Task-based Paradigm

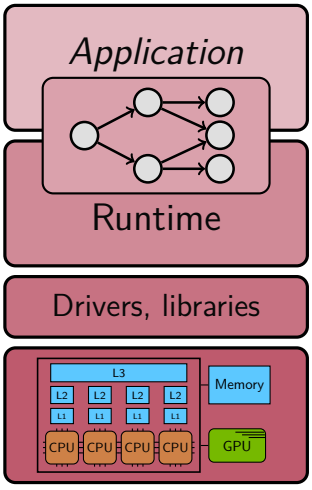
- Applications are presented as a Directed Acyclic Graph (DAG).
  - Nodes are *tasks*, a set of computations.
  - Edges are *dependencies* that ensure the correct workflow of the application.
- Runtime systems enforce the dependencies and schedule the tasks on the computing resources available.



The Task-based Paradigm

- Applications are presented as a Directed Acyclic Graph (DAG).
  - Nodes are *tasks*, a set of computations.
  - Edges are *dependencies* that ensure the correct workflow of the application.
- Runtime systems enforce the dependencies and schedule the tasks on the computing resources available.

⇒ Different models propose different ways for the user to describe an application's DAG to the runtime system.



Sequential Task Flow Model

Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

Sequential Task Flow Model

Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Sequential Task Flow Model

Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

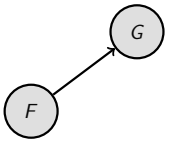
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:



Read after Write on **a**

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

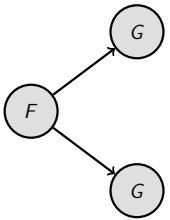
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:



Read after Write on **a**

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

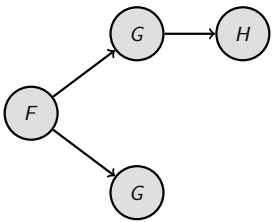
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:



Write after Write on **b**

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

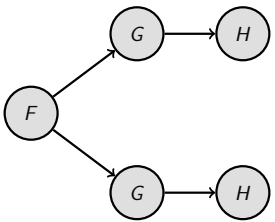
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:



Write after Write on **c**

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

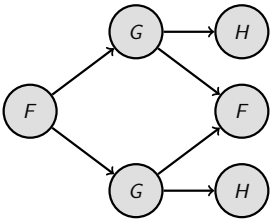
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
```

Resulting DAG:



Write after Read on **a**

- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

Sequential Task Flow Model

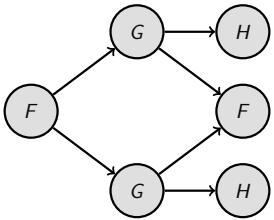
Sequential code:

```
1 F(a)
2 G(a, b)
3 G(a, c)
4 H(b)
5 H(c)
6 F(a)
```

STF code:

```
1 submit(F, a:RW)
2 submit(G, a:R, b:RW)
3 submit(G, a:R, c:RW)
4 submit(H, b:W)
5 submit(H, c:W)
6 submit(F, a:RW)
7 wait_tasks_completion()
```

Resulting DAG:



- The STF model relies on sequential consistency to create data dependencies.
- It provides an intuitive way to express applications.
- It is widely used in state of the art runtime systems (PaRSEC's DTD, OmpSs, OpenMP (> 4.0), etc).

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.

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

Granularity

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

$\Rightarrow$ Steering granularity dynamically ?

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 !

Granularity

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

$\Rightarrow$ Steering granularity dynamically ?

RECURSIVE TASKS

Recursive Tasks in StarPU

Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

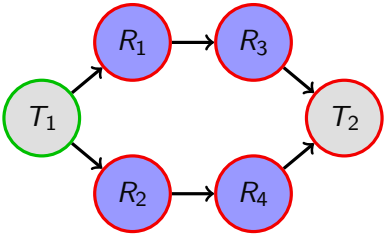
Recursive Tasks in StarPU

Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.



Recursive Tasks in StarPU

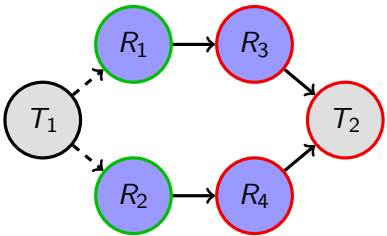
Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

- Recursive task execution:



Recursive Tasks in StarPU

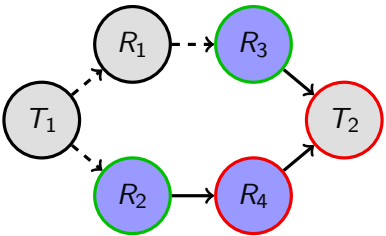
Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

- Recursive task execution:
  - Remain regular task.



Recursive Tasks in StarPU

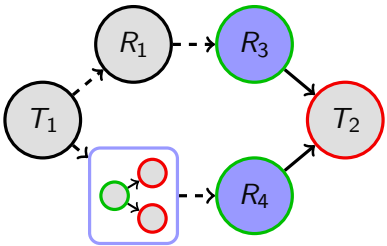
Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

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



Recursive Tasks in StarPU

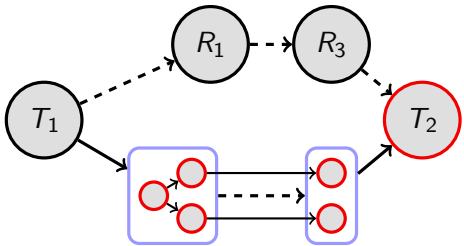
Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.

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



Recursive Tasks in StarPU

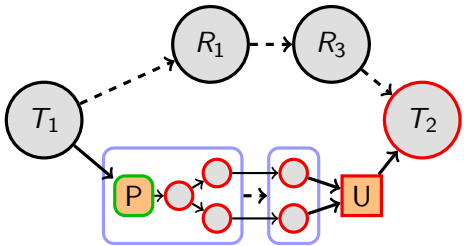
Objectives

- Adapt task implementation *at runtime*.
- No spurious synchronization.

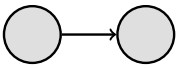
Principles

1. No limit for the hierarchy depth.
2. Fine-grained dependencies.
3. Transparent data management.
   - Automatic data partition.

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



Recursive tasks - State of the Art



Runtime	Fine-grain Dependencies	Automatic data Partition	Heterogeneity
TaskFlow			
PaRSEC			
IRIS			
OmpSs			
StarPU			

Recursive tasks - State of the Art

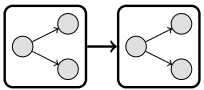


Figure 1: Barrier between parent tasks

Runtime	Fine-grain Dependencies	Automatic data Partition	Heterogeneity
TaskFlow	✗		
PaRSEC	✗		
IRIS	✗		
OmpSs			
StarPU			

Recursive tasks - State of the Art

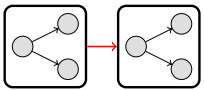


Figure 1: Barrier between parent tasks

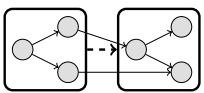


Figure 2: Fine-grain dependencies

Runtime	Fine-grain Dependencies	Automatic data Partition	Heterogeneity
TaskFlow	✗		
PaRSEC	✗		
IRIS	✗		
OmpSs	✓		
StarPU	✓		

Recursive tasks - State of the Art

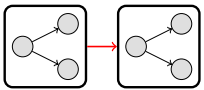


Figure 1: Barrier between parent tasks

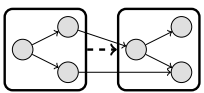


Figure 2: Fine-grain dependencies

Runtime	Fine-grain Dependencies	Automatic data Partition	Heterogeneity
TaskFlow	✗	✗	
PaRSEC	✗	✗	
IRIS	✗	✓	
OmpSs	✓		
StarPU	✓	✓	

Recursive tasks - State of the Art

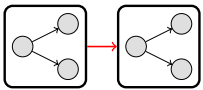


Figure 1: Barrier between parent tasks

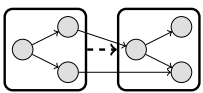


Figure 2: Fine-grain dependencies

Runtime	Fine-grain Dependencies	Automatic data Partition	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
IRIS	✗	✓	✓
OmpSs	✓		✗
StarPU	✓	✓	✓

DYNAMIC TASK GRAPH ADAPTATION

Dynamic task graph adaptation : splitting tasks

Which task should we split?

When do we choose to split task?

Dynamic task graph adaptation : splitting tasks

Which task should we split?

Efficiency VS Completion Time

When do we choose to split task?

Dynamic task graph adaptation : splitting tasks

Which task should we split?

Efficiency VS Completion Time

When do we choose to split task?

Submission, execution, ...

Which task do we split

Exploit informations

Which task do we split

Exploit informations

1. Split efficiency.

Which task do we split

Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Which task do we split

Exploit informations

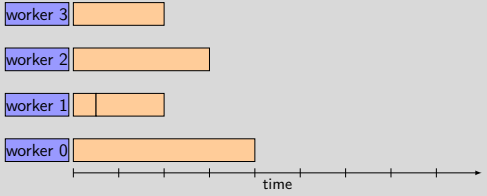
1. **Split efficiency.**
2. Current parallelism on Runtime System.

Which task do we split

Exploit informations

1. **Split efficiency.**
2. Current parallelism on Runtime System.

Split the task - Gantt chart

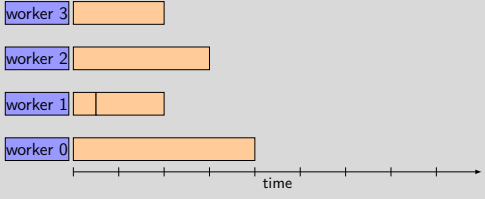


Which task do we split

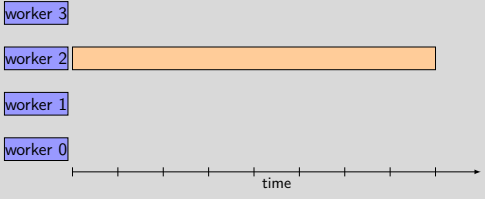
Exploit informations

1. **Split efficiency.**
2. Current parallelism on Runtime System.

Split the task - Gantt chart



Not split the task - Gantt chart

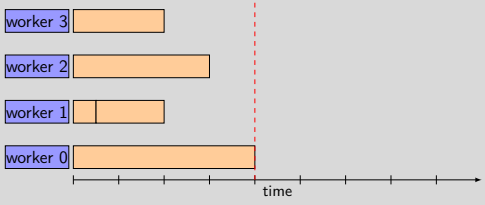


Which task do we split

Exploit informations

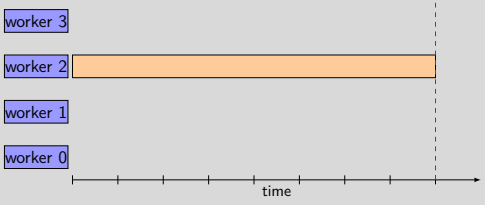
1. **Split efficiency.**
2. Current parallelism on Runtime System.

Split the task - Gantt chart



Completion time: 4 units.

Not split the task - Gantt chart



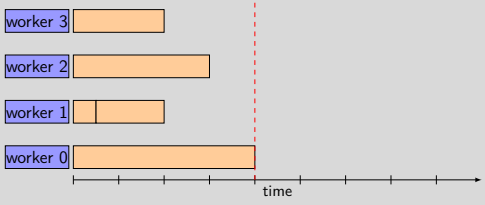
Completion time: 8 units.

Which task do we split

Exploit informations

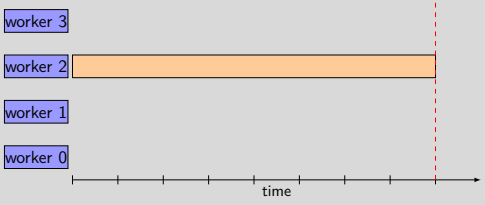
1. **Split efficiency.**
2. Current parallelism on Runtime System.

Split the task - Gantt chart



Completion time: 4 units.
Cumulated time: 11 units.

Not split the task - Gantt chart



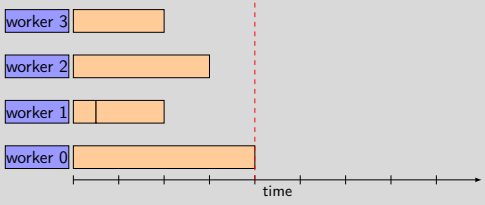
Completion time: 8 units.
Cumulated time: 8 units.

Which task do we split

Exploit informations

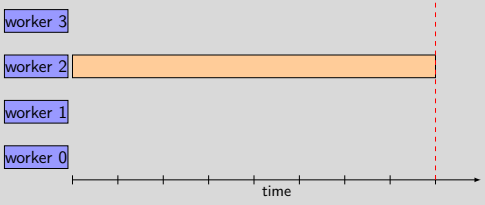
1. **Split efficiency.**
2. Current parallelism on Runtime System.

Split the task - Gantt chart



Completion time: 4 units.
Cumulated time: 11 units.

Not split the task - Gantt chart



Completion time: 8 units.
Cumulated time: 8 units.

Which task do we split

Exploit informations

1. Split efficiency.
2. Current parallelism on Runtime System.

Which task do we split

Exploit informations

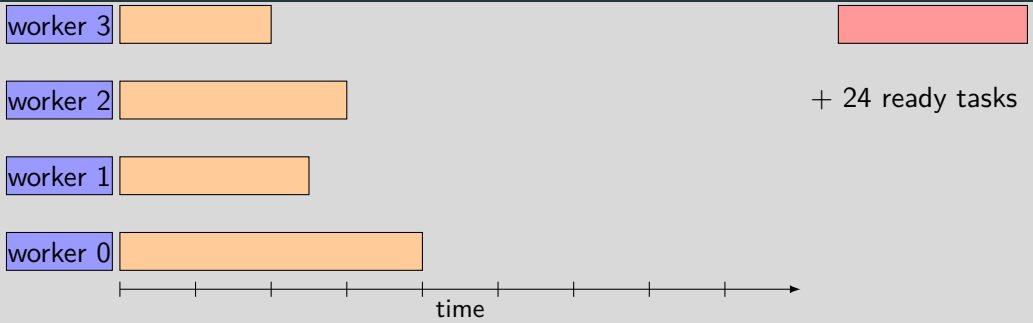
1. Split efficiency.
2. **Current parallelism on Runtime System.**

Which task do we split

Exploit informations

1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 1: Steady State

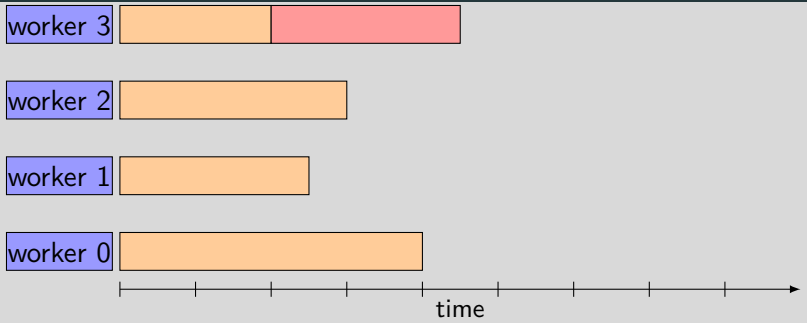


Which task do we split

Exploit informations

1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 1: Steady State

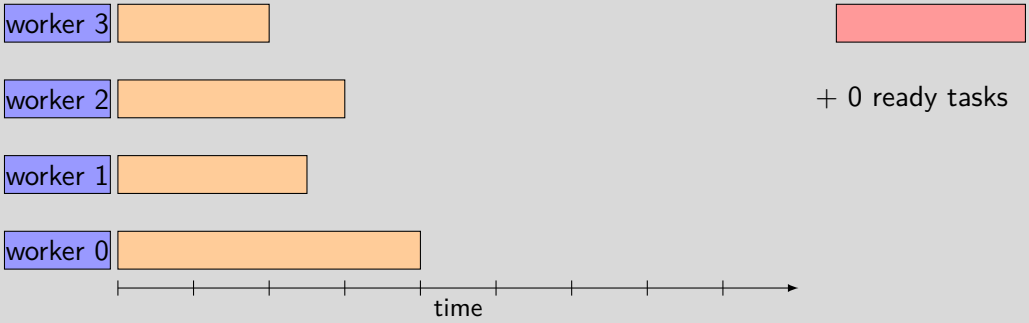


Which task do we split

Exploit informations

1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 2

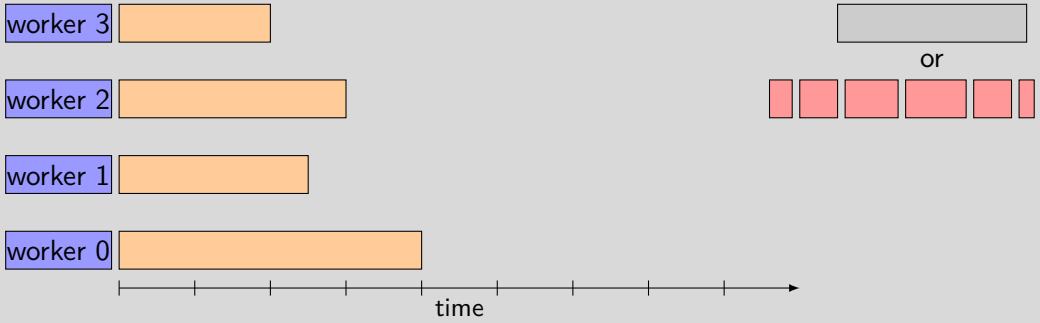


Which task do we split

Exploit informations

1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 2 : Starvation

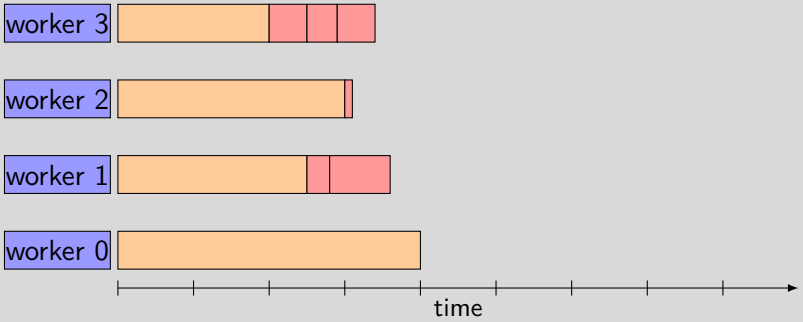


Which task do we split

Exploit informations

1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 2 : Starvation

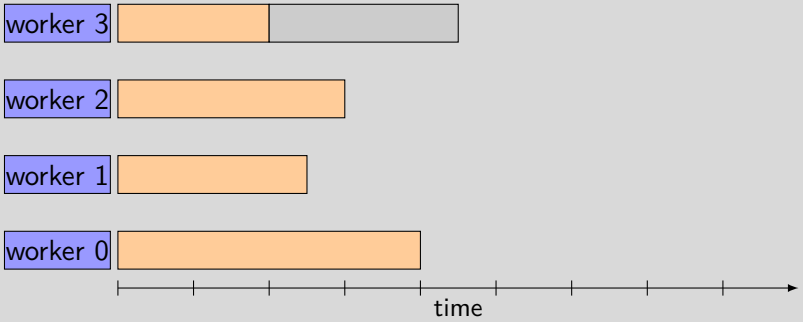


Which task do we split

Exploit informations

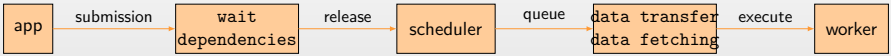
1. Split efficiency.
2. **Current parallelism on Runtime System.**

Situation 2 : Starvation



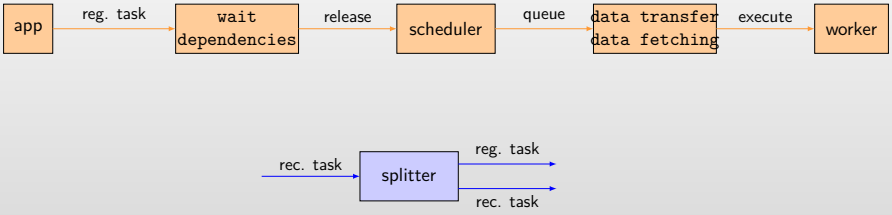
When do we choose to split tasks

Task life path



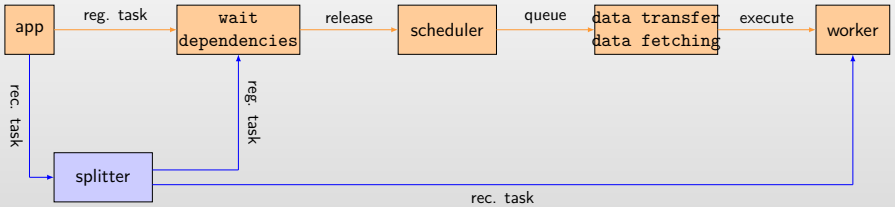
When do we choose to split tasks

Adding the splitter



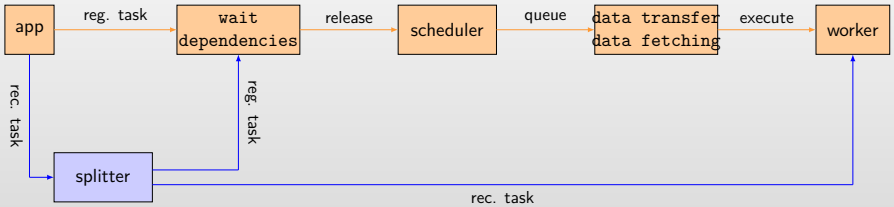
When do we choose to split tasks

Position of the splitter - at submission



When do we choose to split tasks

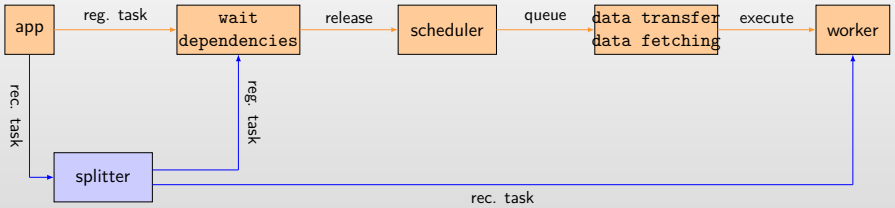
Position of the splitter - at submission



- Easy.

When do we choose to split tasks

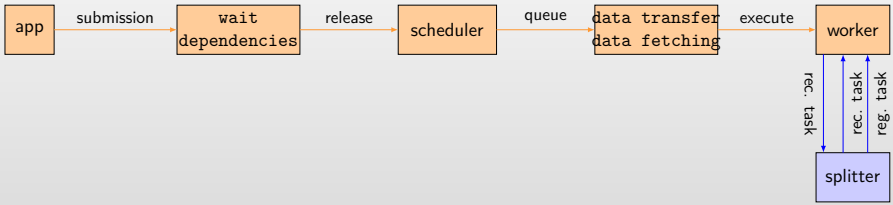
Position of the splitter - at submission



- Easy.
- Lack of information.

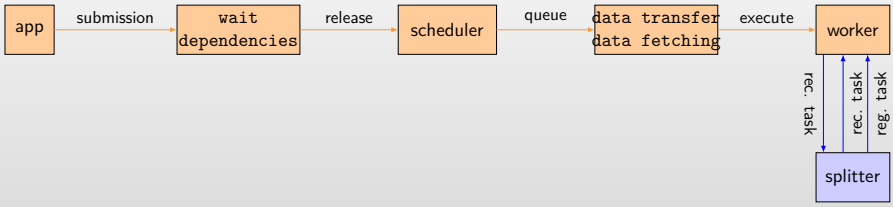
When do we choose to split tasks

Position of the splitter - Execution



When do we choose to split tasks

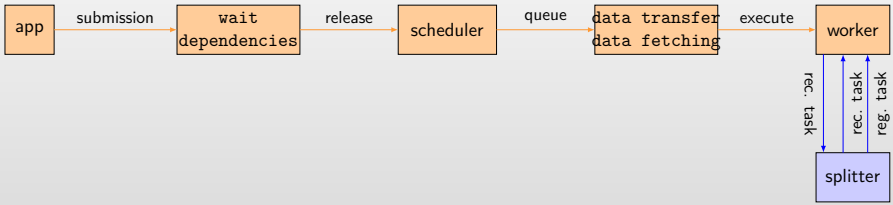
Position of the splitter - Execution



- Runtime information.

When do we choose to split tasks

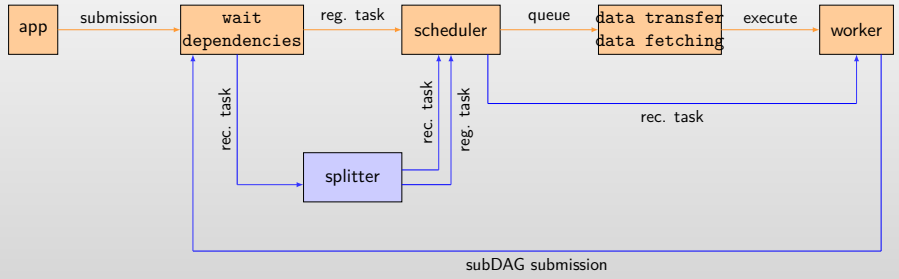
Position of the splitter - Execution



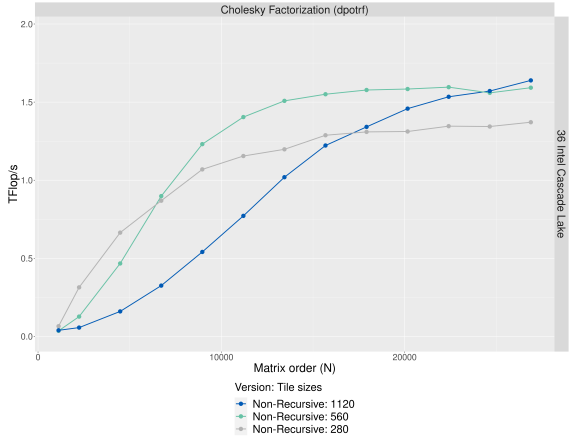
- Runtime information.
- Useless data transfer: cancel decision.

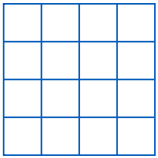
When do we choose to split tasks

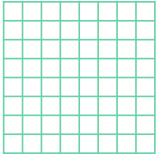
Position of the splitter - trade-off



Results - POTRF (2x18-core Intel Cascade Lake)







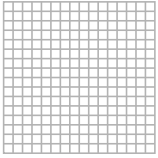
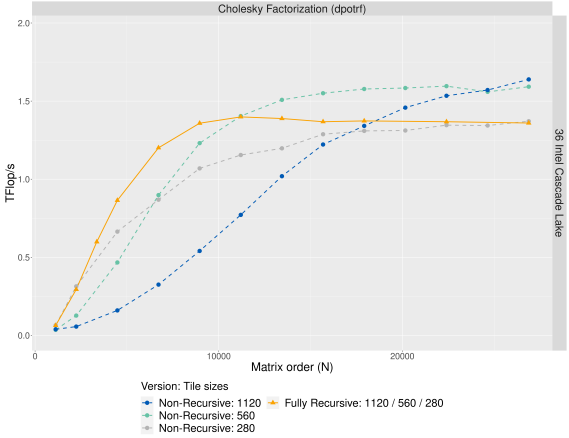


Figure 3: Performance comparison between different Cholesky Factorization versions.

Results - POTRF (2x18-core Intel Cascade Lake)



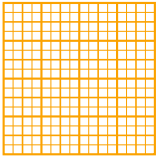
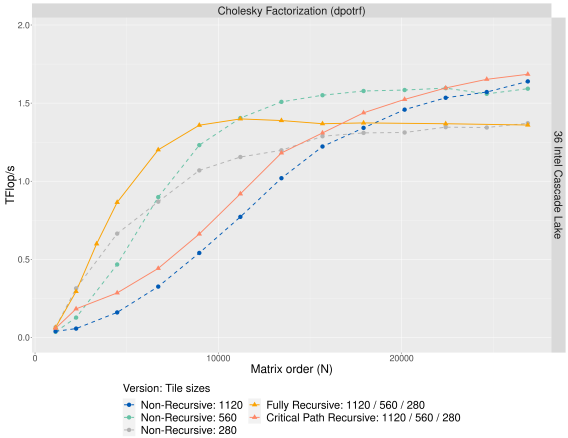


Figure 3: Performance comparison between different Cholesky Factorization versions.

Results - POTRF (2x18-core Intel Cascade Lake)



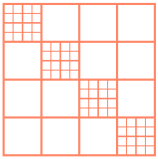
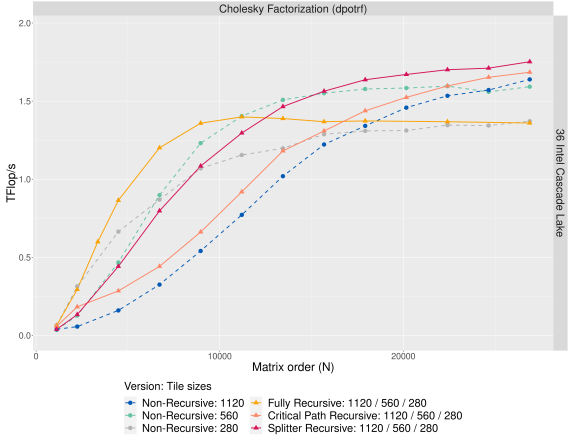


Figure 3: Performance comparison between different Cholesky Factorization versions.

Results - POTRF (2x18-core Intel Cascade Lake)



?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

Figure 3: Performance comparison between different Cholesky Factorization versions.

Results - POTRF (2x18-core Intel Cascade Lake)

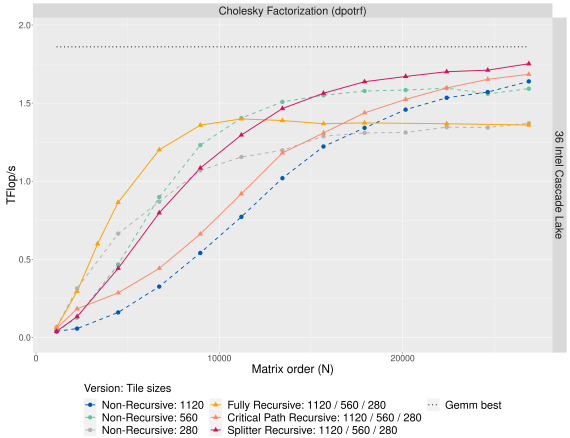


Figure 3: Performance comparison between different Cholesky Factorization versions.

Which task should we split - Heterogeneous case

1. Which type of tasks needs to be split.

Which task should we split - Heterogeneous case

1. Which type of tasks needs to be split.
2. Split the right amount of tasks.

Which task should we split - Heterogeneous case

1. Which type of tasks needs to be split.
2. Split the right amount of tasks.

Which type of tasks needs to be split ?

Objective

$$\min \quad t\_exec \tag{1}$$

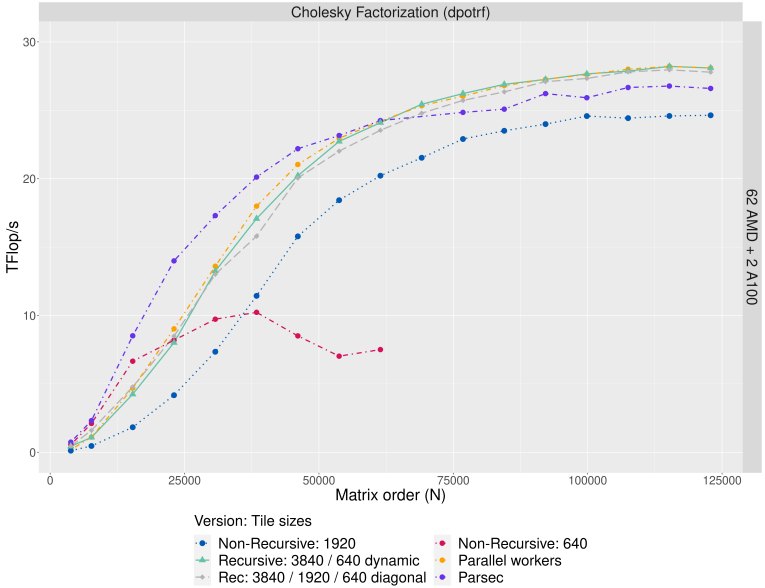
w.r.t.

$$\sum_{t \in \tau} N_t^{big} \cdot Time_t^{gpu} \leq R^{gpus} \cdot t\_exec \tag{2}$$

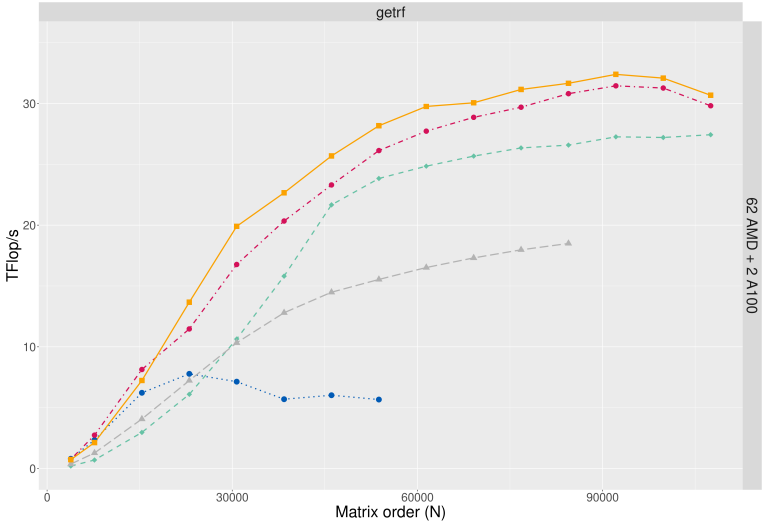
$$\sum_{t \in \tau} N_t^{small} \cdot Time_t^{cpu} \leq R^{cpus} \cdot t\_exec \tag{3}$$

$$N_t^{small} + N_t^{big} = N_t^{total} \qquad \forall\, t \in \tau \tag{4}$$

Results - POTRF (2xNvidia A100 + 2x32-core AMD Zen3 EPYC 7513)



Results - GETRF (2xNvidia A100 + 2x32-core AMD Zen3 EPYC 7513)



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.

Future Work

- Scheduling questions:
  - How should we split tasks ?
- Extend current work:
  - Distributed recursive tasks.