Inria

Programming Heterogeneous Architectures using Hierarchical Tasks

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

March 24th 2023

Task Based Programming

- Task-based programming aims to provide portable frameworks capable of exploiting complex architectures.
- 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 handle scheduling, communications, ...

Task Based Programming

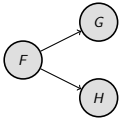
- Task-based programming aims to provide portable frameworks capable of exploiting complex architectures.
- Applications are presented as a Directed Acyclic Graph (DAG).
  - > Nodes are *tasks*, a set of computations.
  - > Edges are *dependencies* that ensure the correct workflow of the application.
- Runtime systems handle scheduling, communications, ...

StarPU

- StarPU rely on the *Sequential Task Flow* (STF) to create its DAGs.
- The STF infers dependencies from the order of submission of the tasks and data access modes.

```
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()
```



Inria

. . . of tasks based programming

- GPUs and CPUs work best on different granularities.
- Some applications are too irregular to fit in a predetermined task-graph.
- Static task graphs limit adaptability during runtime.

. . . of the STF model

- Runtime overhead induced by a large number of non-ready tasks.
- The sequential insertion of tasks can bottleneck the execution of large DAG.

... of tasks based programming

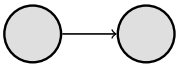
- GPUs and CPUs work best on different granularities.
- Some applications are too irregular to fit in a predetermined task-graph.
- Static task graphs limit adaptability during runtime.

... of the STF model

- Runtime overhead induced by a large number of non-ready tasks.
- The sequential insertion of tasks can bottleneck the execution of large DAG.

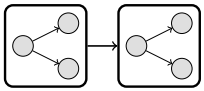
$\Rightarrow$ How to create more dymanic task-graphs?

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:



	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow			
PaRSEC			
OmpSs			
IRIS			

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:



	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
OmpSs			
IRIS			

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:

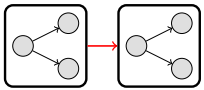


Figure: Barrier between parent tasks

	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
OmpSs			
IRIS			

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:

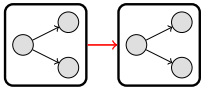
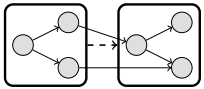


Figure: Barrier between parent tasks



	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
OmpSs	✓	✗	✗
IRIS			

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:

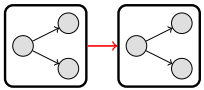


Figure: Barrier between parent tasks

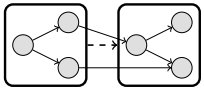


Figure: Fine-grain dependencies

	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
OmpSs	✓	✗	✗
IRIS			

The following runtimes aim at creating more dynamic task-graphs by replacing tasks with an equivalent subgraph:

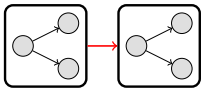


Figure: Barrier between parent tasks

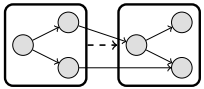


Figure: Fine-grain dependencies

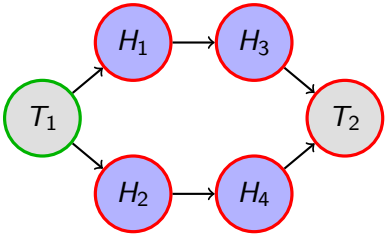
	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
TaskFlow	✗	✗	✓
PaRSEC	✗	✗	✓
OmpSs	✓	✗	✗
IRIS	?	✓	✓

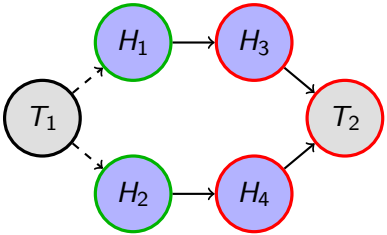
Objectives

- Adapt task granularity to devices
- Reduce the amount of active tasks in StarPU
- Dynamically adapt task implementation at runtime

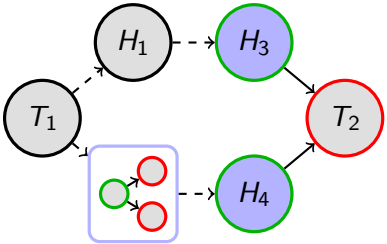
Principles

1. No limit for the hierarchy depth
2. Users simply annotates tasks as hierarchical in the coarse graph
3. Data management is transparent to the programmer
4. Dependencies connect tasks at the finest level possible

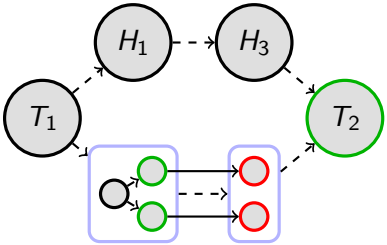

H_1
H_3
T_1
T_2
H_2
H_4


H_1
H_3
T_1
T_2
H_2
H_4

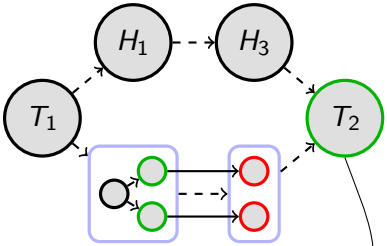
- When executed, a hierarchical task *can* insert a subgraph.



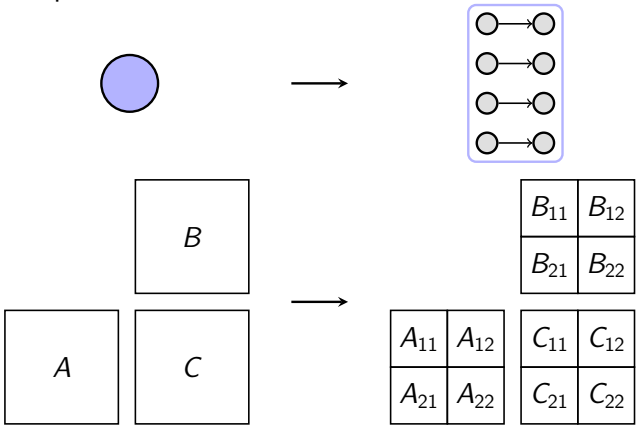
- When executed, a hierarchical task *can* insert a subgraph.
- The dependencies in the subgraphs are infered from the data.



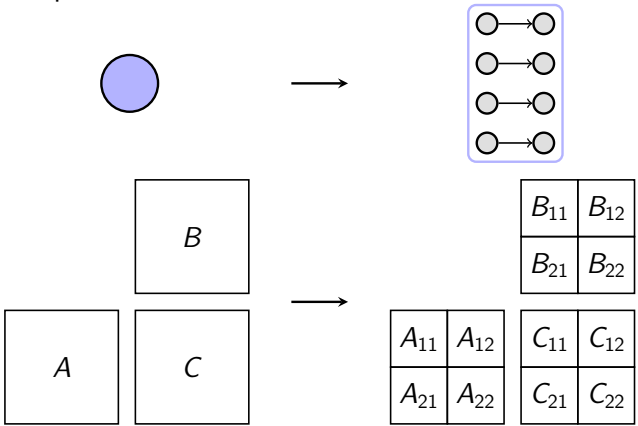
- When executed, a hierarchical task *can* insert a subgraph.
- The dependencies in the subgraphs are infered from the data.



Matrix-matrix multiplication $C = C + A \times B$:



Matrix-matrix multiplication $C = C + A \times B$:



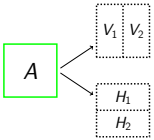
$\Rightarrow$ How to adapt data partitioning to suit hierarchical tasks?


```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)
.
```

- The user describes data partitioning through *plan* operations.

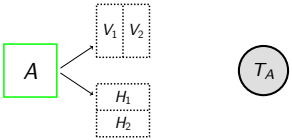

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)
.
```

- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.



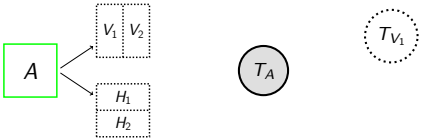

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)
.
```

- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.



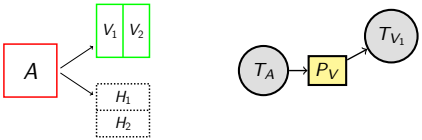

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)

.
```

- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.



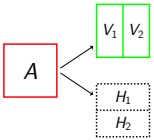

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
.
```

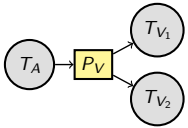
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.




```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
.
```

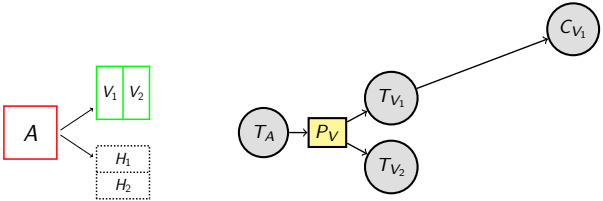
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.





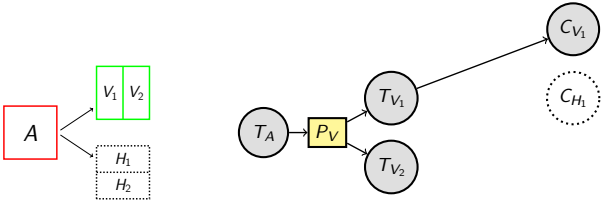

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
.
```

- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.



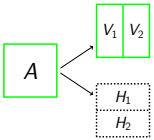

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

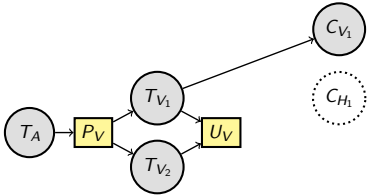
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.




```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

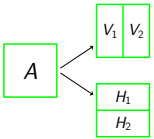
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.

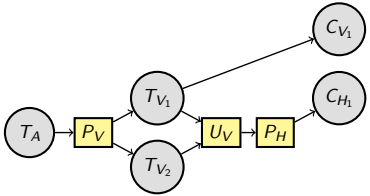





```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

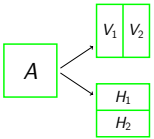
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.

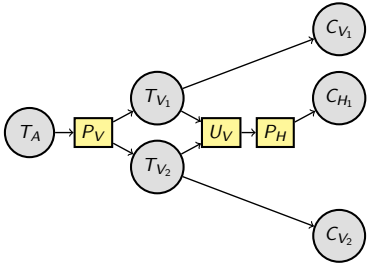





```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

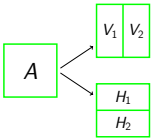
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.

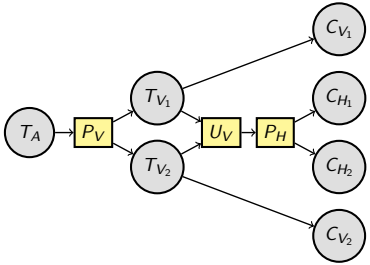





```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

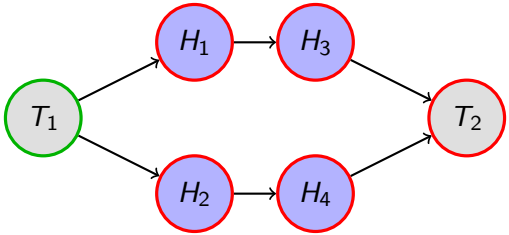
- The user describes data partitioning through *plan* operations.
- Multiple *plans* for the same data piece can coexist.
- Partitioning tasks are inserted to ensure that each task will use a valid data.





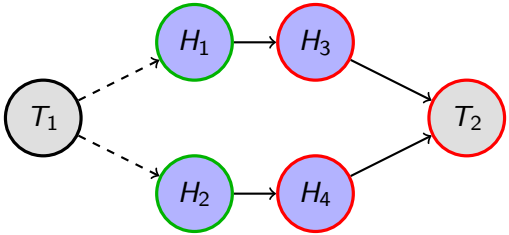

```
[Data registration]
register(A)
partition_plan(A, vfilter, V)
partition_plan(A, hfilter, H)

[Modification]
submit(T, A:RW)
for (i = 1 to NPARTS)
    submit(T, V[i]:RW)
[Verification]
for (i = 1 to NPARTS)
    submit(C, V[i]:R)
    submit(C, H[i]:R)

wait_tasks_completion()
```

Correctness of a Hierarchical DAG

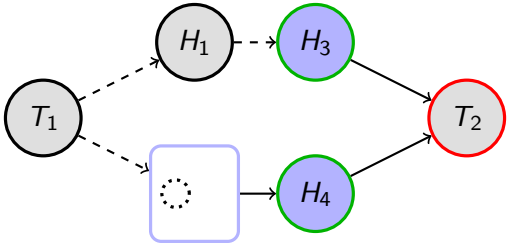


Inria

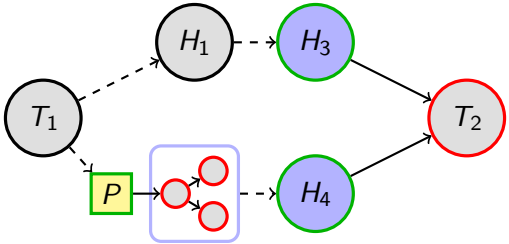
Correctness of a Hierarchical DAG



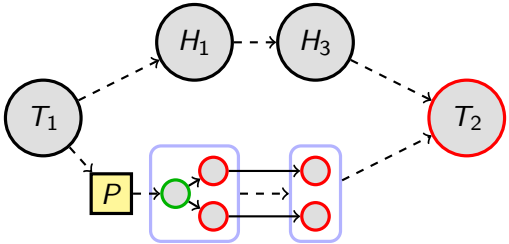
- Partitioning tasks are added, if needed, at the submission of a subtask.



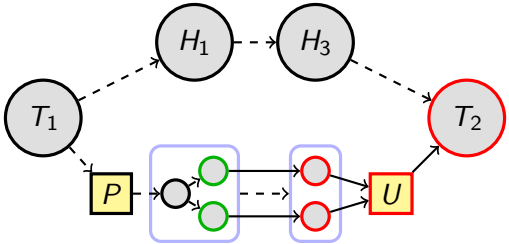
- Partitioning tasks are added, if needed, at the submission of a subtask.



- Partitioning tasks are added, if needed, at the submission of a subtask.



- Partitioning tasks are added, if needed, at the submission of a subtask.
- Unpartitioning tasks are added, if needed, before a regular task. They enforce the correctness of the DAG.



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

- 2x 16-core Skylake Intel Xeon Gold 6142 @ 2.6 GHz
- 2 NVIDIA V100 (16GB)
- 384 GB (12 GB/core) (@2666 MHz)

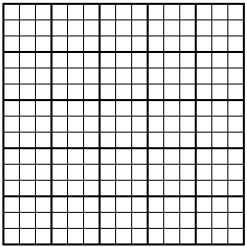


Figure: Full matrix partitioning.

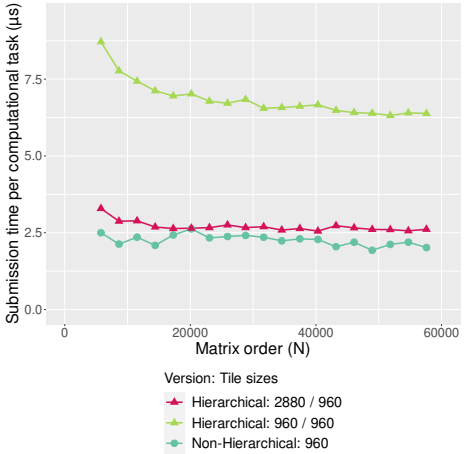


Figure: Submission cost of computational tasks for the matrix-matrix multiplication kernel.

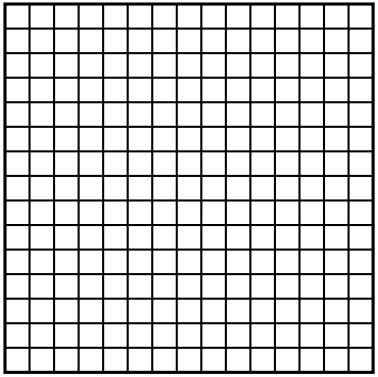


Figure: Tile size of 960.

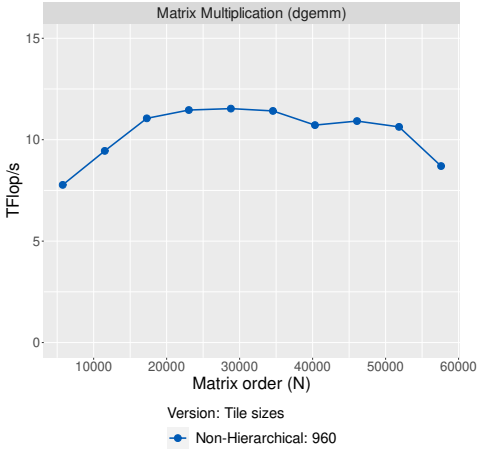


Figure: Matrix-matrix multiplication kernel with a fixed percentage of hierarchical tasks on Intel-V100.

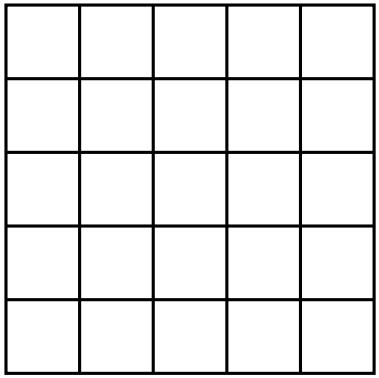


Figure: Tile size of 2880.

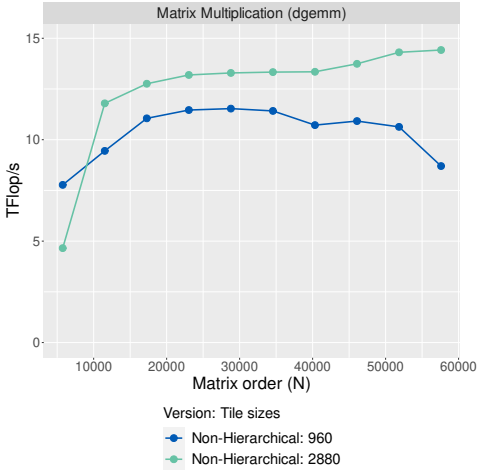


Figure: Matrix-matrix multiplication kernel with a fixed percentage of hierarchical tasks on INTEL-V100.

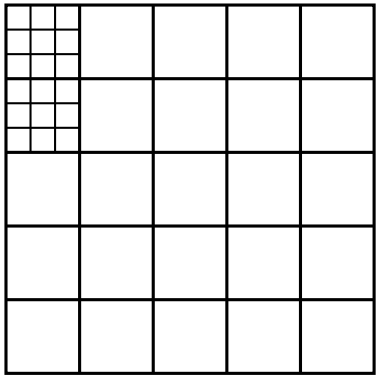


Figure: 10% recursive matrix partitioning.

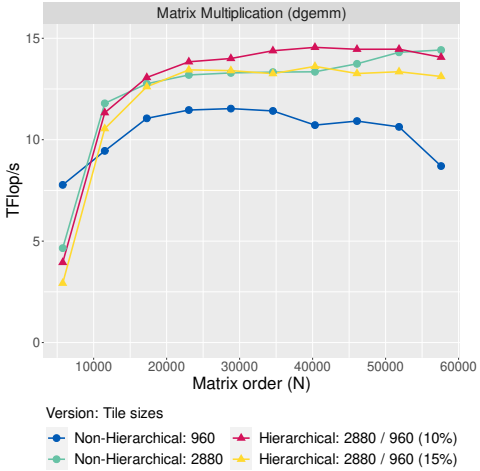


Figure: Matrix-matrix multiplication kernel with a fixed percentage of hierarchical tasks on INTEL-V100.

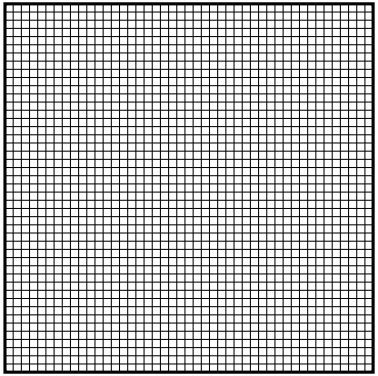


Figure: Diagonal matrix partitioning.

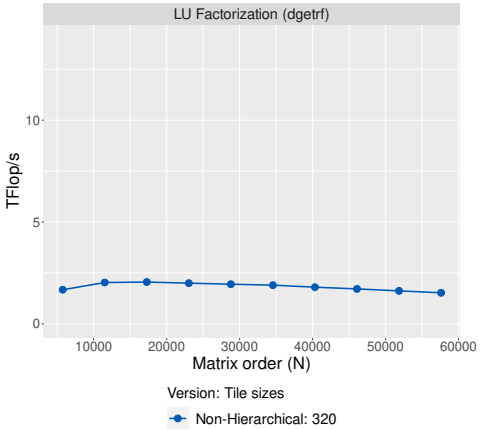


Figure: LU factorization kernel with a diagonal repartition of hierarchical tasks on INTEL-V100.

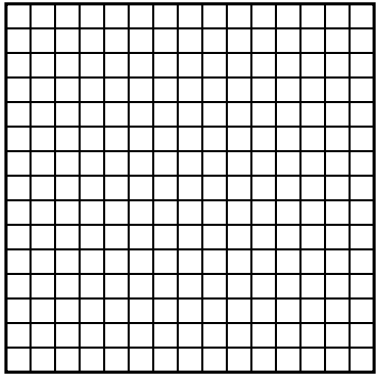


Figure: Diagonal matrix partitioning.

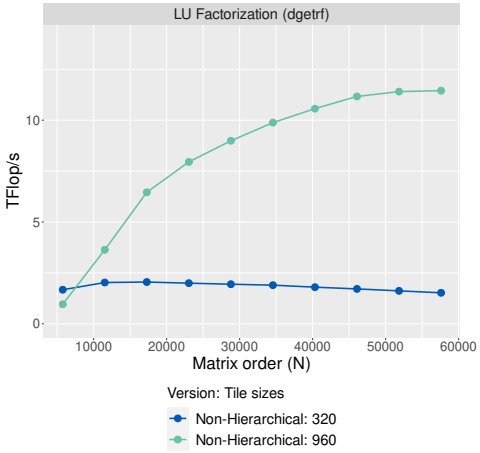


Figure: LU factorization kernel with a diagonal repartition of hierarchical tasks on INTEL-V100.

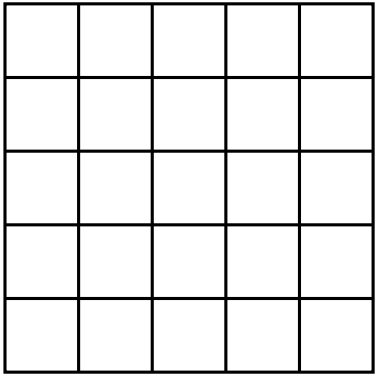


Figure: Diagonal matrix partitioning.

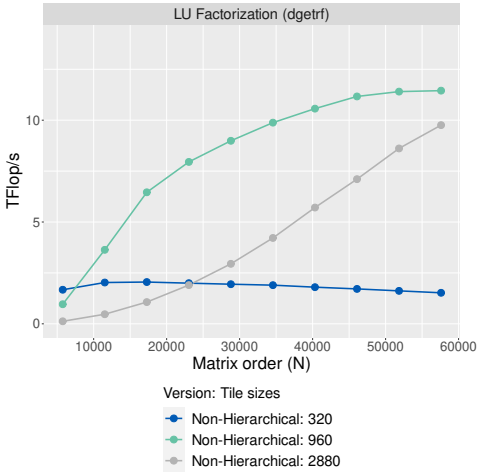


Figure: LU factorization kernel with a diagonal repartition of hierarchical tasks on INTEL-V100.

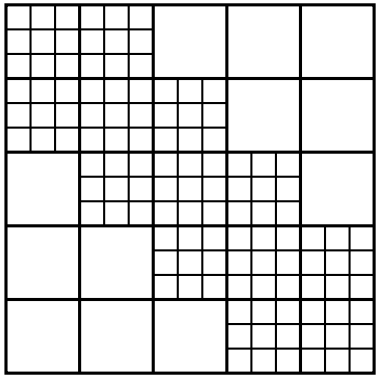


Figure: Diagonal matrix partitioning.

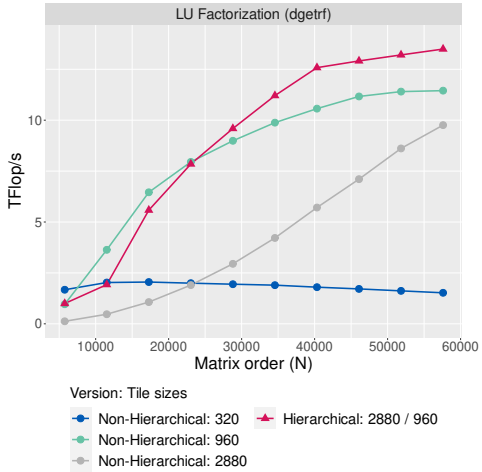


Figure: LU factorization kernel with a diagonal repartition of hierarchical tasks on INTEL-V100.

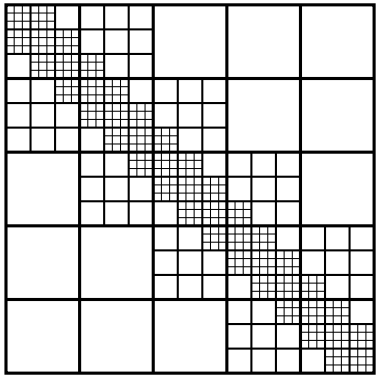


Figure: Diagonal matrix partitioning.

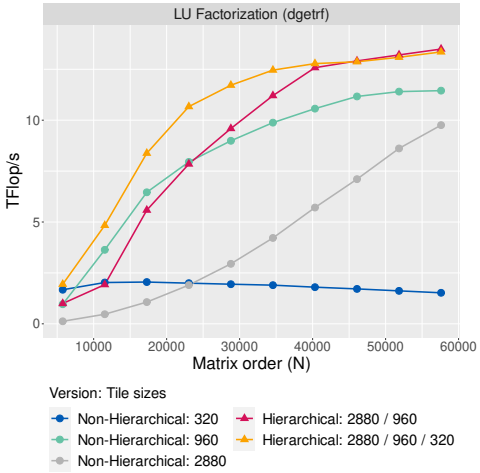


Figure: LU factorization kernel with a diagonal repartition of hierarchical tasks on INTEL-V100.

Benchmarks - Cholesky Factorization

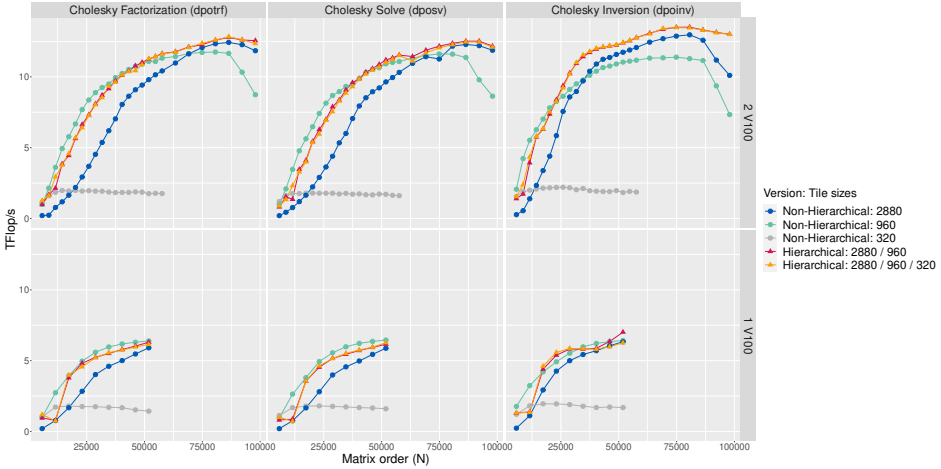
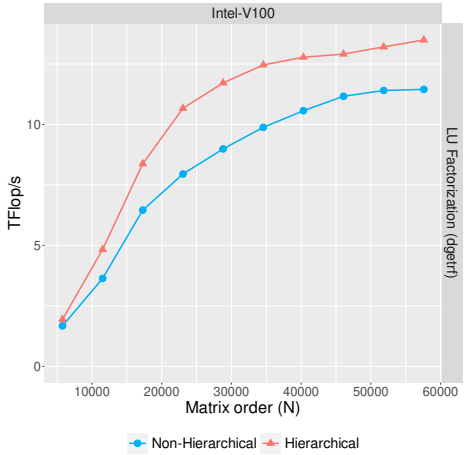
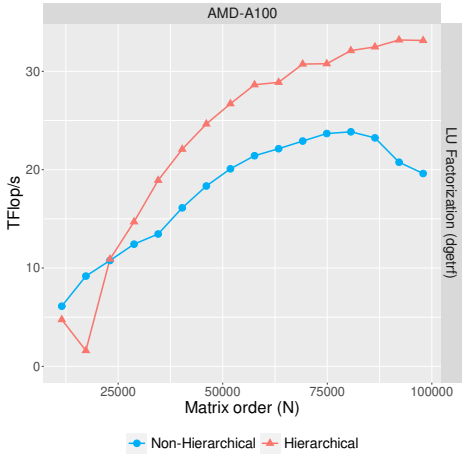


Figure: Cholesky type operations (DPOTRF, DPOSV, DPOINV) kernel with diagonal distribution of the hierarchical tasks on INTEL-V100.

Benchmarks - Best performance - LU No Pivoting





- Hierarchical tasks can insert a subgraph at runtime, resulting in a more dynamic DAG.
- Data management is handled automatically and contributes to the correctness of hierarchical DAGs.

Future Work

- Scheduling questions:
  - When should we insert a subgraph ?
  - Where should we execute it ?
  - Using which implementation ?
- Testing with applications benefitting more from dynamic task graphs (sparse solvers, low rank approximation, . . .)

- Hierarchical tasks can insert a subgraph at runtime, resulting in a more dynamic DAG.
- Data management is handled automatically and contributes to the correctness of hierarchical DAGs.

Future Work

- Scheduling questions:
  - When should we insert a subgraph ?
  - Where should we execute it ?
  - Using which implementation ?
- Testing with applications benefitting more from dynamic task graphs (sparse solvers, low rank approximation, . . .)

And Gwenolé is currently looking for a postdoc position :D