



**HAL**  
open science

## **Batching the tasks of the LU factorization with partial pivoting on top of runtime systems**

Alycia Lisito, Mathieu Faverge, Matthieu Kuhn, Florent Pruvost, Pierre Ramet

### ► **To cite this version:**

Alycia Lisito, Mathieu Faverge, Matthieu Kuhn, Florent Pruvost, Pierre Ramet. Batching the tasks of the LU factorization with partial pivoting on top of runtime systems. COMPAS 2025 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jun 2025, Bordeaux, France. <hal-05148627>

**HAL Id: hal-05148627**

**<https://inria.hal.science/hal-05148627v1>**

Submitted on 7 Jul 2025

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

EVIDEN

*Inria*

# Batching the tasks of the LU factorization with partial pivoting on top of runtime systems

**A. Lisito**, M. Faverge, M. Kuhn, F. Pruvost, P. Ramet

Compas 2025

Eviden, Centre Inria de l'université de Bordeaux

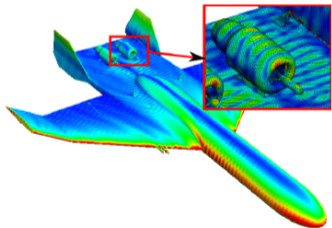
© Eviden SAS

an atos business

# The LU factorization with partial pivoting

## Where is it used ?

- ▶ Numerical simulation
  - ▶ Maxwell equations in the frequency domain via the Boundary Element Method
- ▶ Evaluate supercomputers
  - ▶ The TOP500 based on the HPL Benchmark performance (solves  $Ax = b$  with  $A = PLU$ )



# LU factorization with Partial Pivoting (GETRF)

## Problems

- ▶ Few implementations, most are private and implement HPL not just LU factorization with partial pivoting
- ▶ Supercomputers have thousands of nodes, million cores and are heterogeneous  
→ need to be able to exploit efficiently all the resources
- ▶ Many constructors (Intel, Nvidia, AMD, ...) = many architectures  
→ need to be portable

# LU factorization with Partial Pivoting (GETRF)

## Problems

- ▶ Few implementations, most are private and implement HPL not just LU factorization with partial pivoting
- ▶ Supercomputers have thousands of nodes, million cores and are heterogeneous  
→ need to be able to exploit efficiently all the resources
- ▶ Many constructors (Intel, Nvidia, AMD, ...) = many architectures  
→ need to be portable

## Our goal:

**Implement an efficient scalable and portable (and heterogeneous) task based LU factorization with partial pivoting on top of runtime systems**

# LU factorization with Partial Pivoting (GETRF)

## Problems

- ▶ Few implementations, most are private and implement HPL not just LU factorization with partial pivoting
- ▶ Supercomputers have thousands of nodes, million cores and are heterogeneous  
→ need to be able to exploit efficiently all the resources
- ▶ Many constructors (Intel, Nvidia, AMD, ...) = many architectures  
→ need to be portable

## Our goal:

**Implement an efficient scalable and portable (and heterogeneous) task based LU factorization with partial pivoting on top of runtime systems**

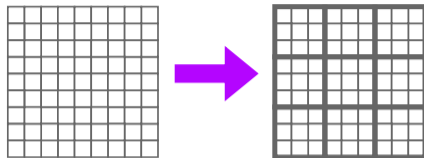
## STARPU and CHAMELEON

- 🌈 Parallel task based runtime system
- 🌈 Task based parallel dense linear algebra library
- 🌈 Tested and validated on large heterogeneous scale
- 🌈 Efficient on all types of architecture

# Task based algorithm

## How does it work ?

- ▶ The matrix is split into blocks to reduce the size of the problem
- ▶ The algorithms are adapted

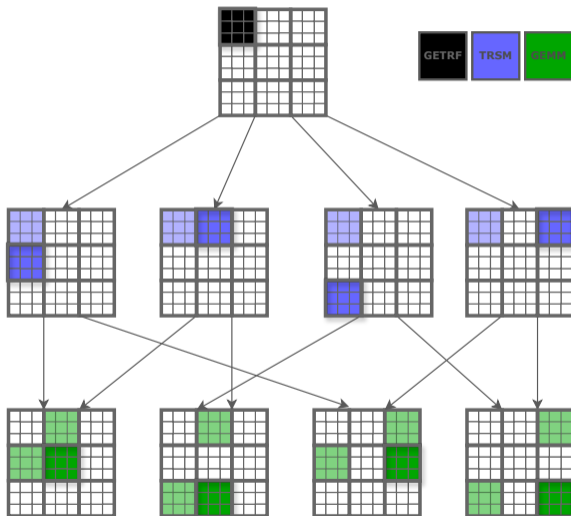


# Task based algorithm

## How does it work ?

- ▶ The matrix is split into blocks to reduce the size of the problem
- ▶ The algorithms are adapted
- ▶ Each operation on a block corresponds to one task
- ▶ Dependencies between the tasks (block in read and / or write mode)

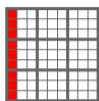
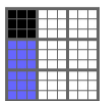
## LU factorisation without partial pivoting



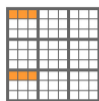
# The LU factorization algorithm

## The panel operations

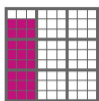
No piv



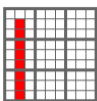
GETRF



TRSM



UPDATE

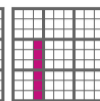
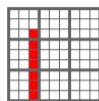


MAX

Partial piv



SWAP



Without partial pivoting

☹ Not numerically stable

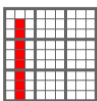
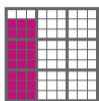
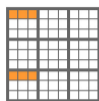
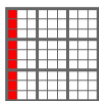
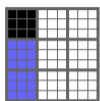
With partial pivoting

☺ Numerically stable

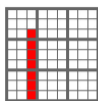
# The LU factorization algorithm

## The panel operations

No piv



Partial piv



## Without partial pivoting

- ☞ Not numerically stable

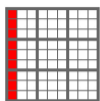
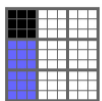
## With partial pivoting

- ☞ Numerically stable
- ☞ High number of tasks → can overflow the runtime

# The LU factorization algorithm

## The panel operations

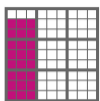
No piv



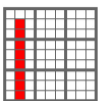
GETRF



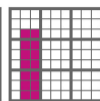
TRSM



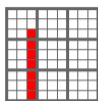
UPDATE



Partial piv



MAX



SWAP



## Without partial pivoting

- ☹ Not numerically stable

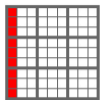
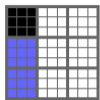
## With partial pivoting

- ☺ Numerically stable
- ☹ High number of tasks → can overflow the runtime
- ☹ Very small tasks → more than 80% of the time in the runtime overhead

# The LU factorization algorithm

## The panel operations

No piv



GETRF



TRSM

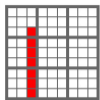


UPDATE



MAX

Partial piv



SWAP



## Without partial pivoting

- ⌚ Not numerically stable

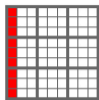
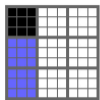
## With partial pivoting

- 🌈 Numerically stable
- ⌚ High number of tasks → can overflow the runtime
- ⌚ Very small tasks → more than 80% of the time in the runtime overhead
- ⌚ Low granularity tasks → update operation with Level 1 BLAS and Level 2 BLAS
- ⌚ Many reductions → lots of synchronisation in the critical path

# The LU factorization algorithm

## The panel operations

No piv



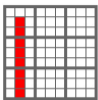
GETRF



TRSM



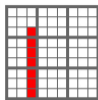
UPDATE



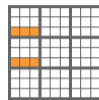
MAX



SWAP



Partial piv



## Without partial pivoting

- ☞ Not numerically stable

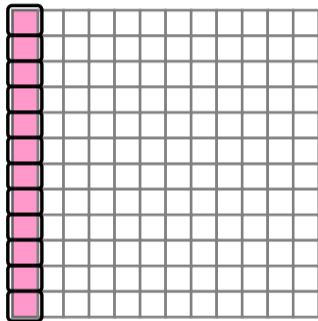
## With partial pivoting

- ☞ Numerically stable
- ☞ **High number of tasks** → can overflow the runtime
- ☞ **Very small tasks** → more than 80% of the time in the runtime overhead
- ☞ Low granularity tasks → update operation with Level 1 BLAS and Level 2 BLAS
- ☞ Many reductions → lots of synchronisation in the critical path

# Batching the tasks

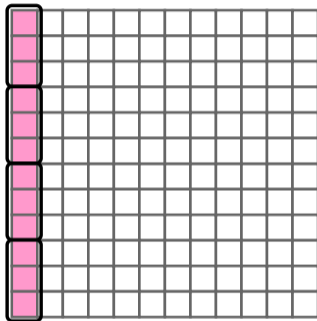
## Shared memory case

No batch / batch = 1



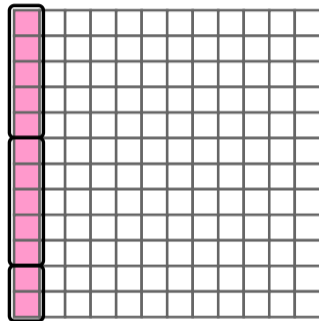
- 12 small tasks
- 80% of the time in the runtime overhead

batch = 3



- 4 tasks 3 times bigger
- less runtime overhead

batch = 5

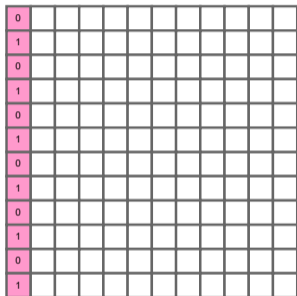


- 3 tasks 4 times bigger
- almost no runtime overhead

# Batching the tasks

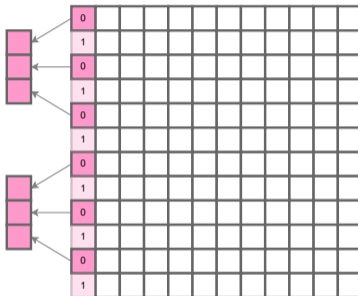
## Distributed memory case

Blocks distributed on 0 and 1



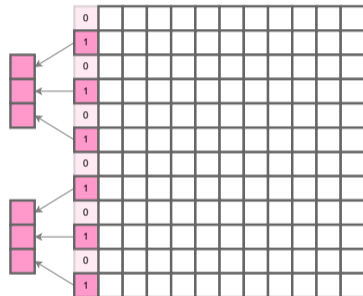
🌈 Blocks distributed on 2 processes

Batch 3 on 0



🌈 2 tasks on process 0

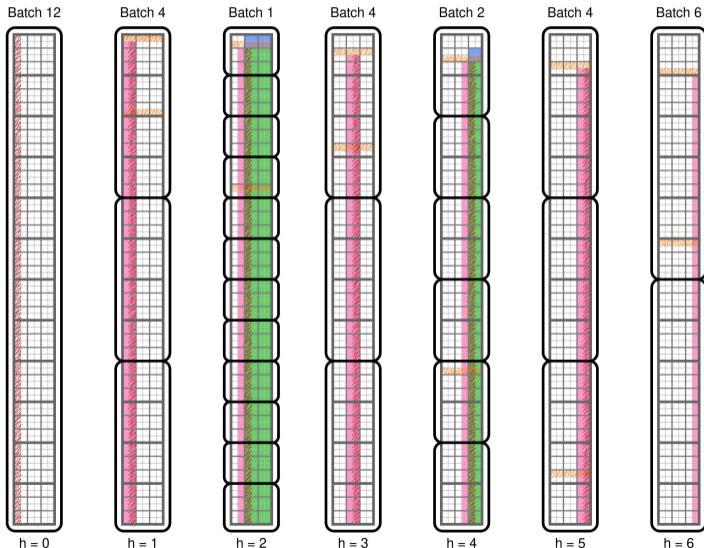
Batch 3 on 1



🌈 2 tasks on process 1

# Batching the tasks

## Adaptative batch size



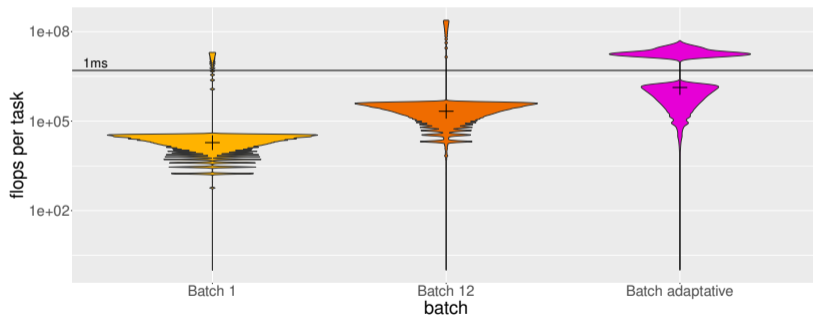
- ☀ Computes the size of the batch at each column  $h$  iteration
- ▶ The minimum of flops per task  $flops_{min}$
- ▶ The number of blocks in the factorisation  $nb_{left}$
- ▶ The number of flops of the factorisation  $flops$
- ▶ The maximum size of batch possible  $b_{max} = \min(MAX, nb_{left})$

$$batch_{size} = \min \left( \left[ \frac{nb_{left}}{\min(\max(\frac{flops_{min}}{flops}, 1), b_{max})} \right], b_{max} \right)$$

# Blocking and batching

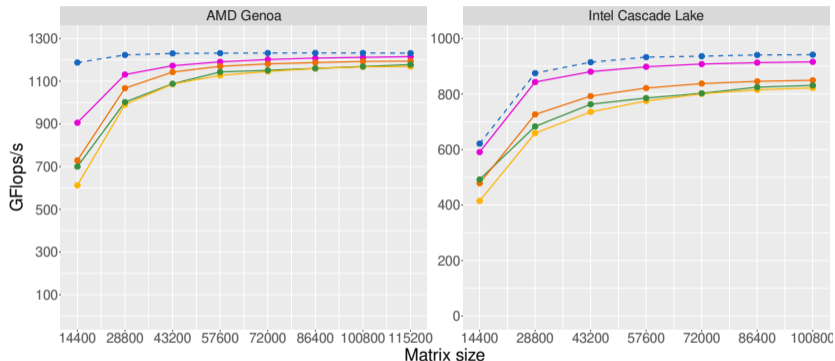
Matrix size  $115200 \times 115200$  with *blocksize* = 576

	batch 1	batch 12	batch adaptative
Tasks nbr	11 601 100	1 023 536	461 996
flops/task mean	330 003	3 740 363	8 286 644
flops/task median	19 008	214 272	1 339 642



# Blocking and batching

AMD Genoa - 24 cores / Intel Cascade Lake - 18 cores



## Variant

- Chameleon GETRF NoPiv
- Dplasma GETRF PPiv
- Chameleon GETRF PPiv batch 1
- Chameleon GETRF PPiv batch 12
- Chameleon GETRF PPiv batch adaptive

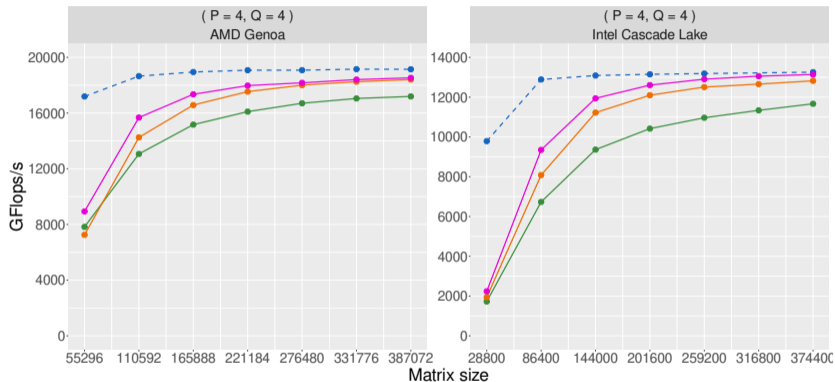
## LU algorithm

- PPiv
- NoPiv

- ▶ GETRF\_NOPIV : reaches a peak at 1200 *GFlops/s* and 940 *GFlops/s*
- ▶ DPLASMA : same as batch 1
- 🌈 batch 1: AMD 95% and Intel 87% of GETRF\_NOPIV
- 🌈 batch 12: AMD 97% and Intel 90% of GETRF\_NOPIV
- 🌈 batch adaptive: AMD 99% and Intel 97% of GETRF\_NOPIV

# All Reduce

8 AMD Genoa - 24 cores / 8 Intel Cascade Lake - 18 cores - 2 MPI processes per node



## Variant

- Chameleon GETRF NoPiv
- Dplasma GETRF PPiv
- Chameleon GETRF PPiv batch 12
- Chameleon GETRF PPiv batch adaptive

## LU algorithm

- PPiv
- NoPiv

- GETRF\_NOPIV : speedup of 15.5 on AMD and 14 on Intel out of 16
- DPLASMA : similar to batch 1
- batch 12: AMD 96% and Intel 97% of GETRF\_NOPIV
- batch adaptive: AMD 97% and Intel 99% of GETRF\_NOPIV
- batch adaptive: speedup of 15 on AMD and 14 on Intel out of 16

# Conclusion

## Contributions

- 🌈 Scalable and portable adaptative batching mechanism

## Future work

- ▶ Batching at the runtime level
- ▶ Manage batching with recursive tasks of STARPU
- ▶ Manage batching in heterogeneous context