Task based parallelization of recursive linear algebra routines using Kaapi

Clément PERNET

joint work with Jean-Guillaume DUMAS and Ziad SULTAN

Université Grenoble Alpes, LJK-CASYS

January 20, 2017

Journée Runtime, Paris.

Supported by OpenDreamKit Horizon 2020 European Research Infrastructures project (# 676541)

High performance algebraic computing

Domain of computation

- $\mathbb{Z}, \mathbb{Q}$: variable size, multi-precision
- $\mathbb{Z}_p$, GF(p^k): fixed size, specific arithmetic

Common belief : Slow

- terrible complexities,
- no need for *all the precision*

Example (Linear System solving over $\mathbb{Q}$)

Method	Complexity
Naive Gauss Elim over $\mathbb{Q}$	$\mathcal{O}\left(2^n\right)$
Gauss mod det	$\mathcal{O}\left(n^5\right)$
Gauss mod p + CRT	$\mathcal{O}\left(n^4\right), \mathcal{O}\left(n^{\omega+1}\right)$
p-adic Lifting	$\mathcal{O}\left(n^3\right), \mathcal{O}^{\sim}\left(n^{\omega}\right)$

And fast software: LU over $(\mathbb{Z}/65521\mathbb{Z})^{5000\times 5000}$ in 3.8s (21.8Gfops on 1 Haswell core)

Gaussian elimination in computer algebra

Applications

Algebraic cryptanalysis: RSA, DLP $\Rightarrow$LinSys, Krylov, $\mathbb{F}_q$

Comp. number theory: modular forms databases: Echelon over $\mathbb{F}_q$

Exact mixed-integer linear programming: $\Rightarrow$LinSys over $\mathbb{Q}$

Formal proof: Sums of squares $\Rightarrow$Cholesky over $\mathbb{Q}$

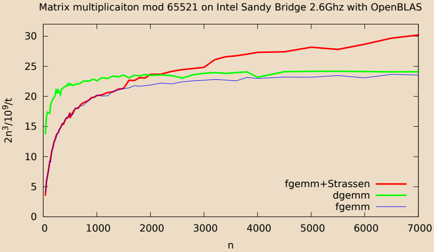
HPC building block

- Dense linear algebra over $\mathbb{Z}/p\mathbb{Z}$ $\log_2 p \approx 20 - 30$ bits
- MatlMul (`fgemm`) and GaussElim (`PLUQ`)
  - triangular decomposition PLUQ (for LinSys, Det)
  - linear dependencies (Krylov, Grobner basis)

FFLAS-FFPACK library

FFLAS-FFPACK features

- High performance implementation of basic linear algebra routines over word size prime fields
- Exact alternative to the numerical BLAS library
- Exact triangularization, Sys. solving, Det, Inv., CharPoly



Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  $\Rightarrow$Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting

  numeric: ensuring numerical stability
  exact: able to reveal rank profile

Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  ⇒Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting

  numeric: ensuring numerical stability
  exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)

Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  ⇒Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  - numeric: ensuring numerical stability
  - exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
- Modular reductions / Strassen's algorithm } efficiency increases with the granularity
  ⇒tradeoff between total work and fine granularity

Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  ⇒Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  numeric: ensuring numerical stability
  exact: able to reveal rank profile

Specificities

- Recursive tasks (vs block iterative in numeric)
- Modular reductions, Strassen's algorithm } efficiency increases with the granularity
  ⇒tradeoff between total work and fine granularity
- Pivoting strategies : no stability constraints, but rank profiles

Exact vs numerical Gaussian elimination

Similarities

- Reduction to `gemm` kernel (Matrix Multiplication)
  ⇒Blocking: slab/tiled, iterative/recursive
- Parallel blocking is constrained by pivoting
  numeric: ensuring numerical stability
  exact: able to reveal rank profile

Specificities

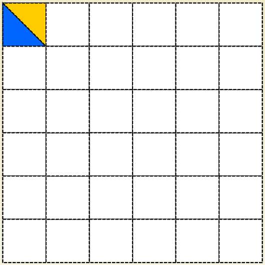
- Recursive tasks (vs block iterative in numeric)
- Modular reductions / Strassen's algorithm } efficiency increases with the granularity
  ⇒tradeoff between total work and fine granularity
- Pivoting strategies : no stability constraints, but rank profiles
- Rank deficienices:
  - blocks have unpredictable size (⇒and positions)
  - unbalanced task load

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive



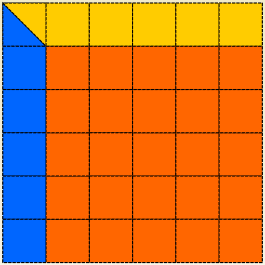
`getrf`: $A \to L, U$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

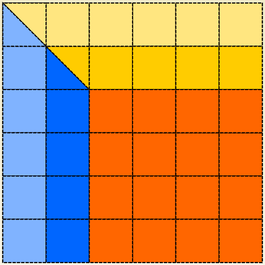


`trsm`: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

`gemm`: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative | Slab Recursive | Tiled Recursive



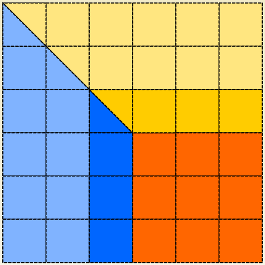
getrf: $A \rightarrow L, U$
trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive



getrf: $A \to L, U$

trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

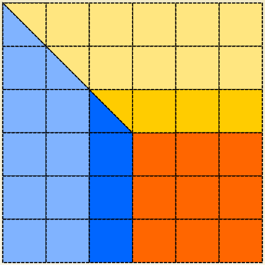
gemm: $C \leftarrow C - A \times B$

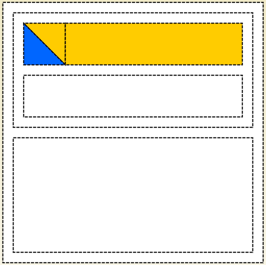
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

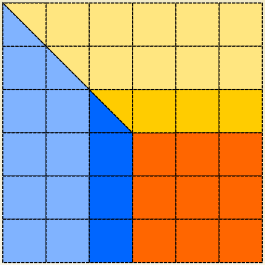




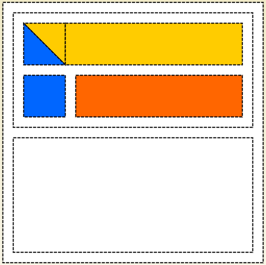
getrf: $A \to L, U$

Block algorithms

Tiled Iterative



Slab Recursive



Tiled Recursive

trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

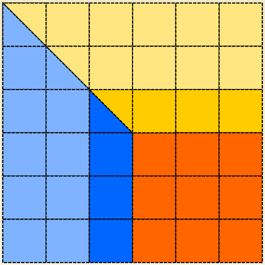
gemm: $C \leftarrow C - A \times B$

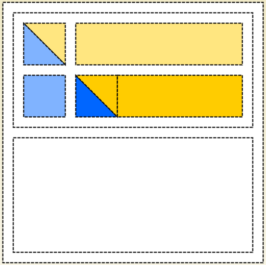
Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

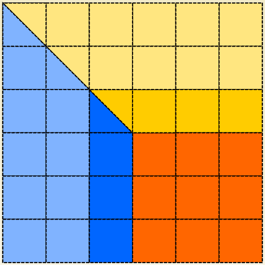




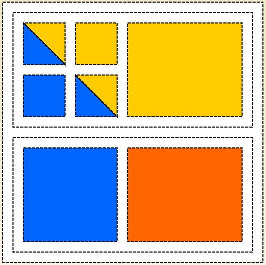
getrf: $A \to L, U$

Block algorithms

Tiled Iterative



Slab Recursive



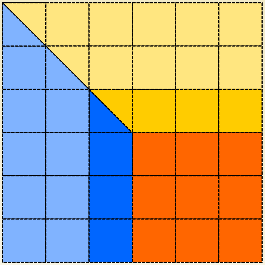
Tiled Recursive

`trsm`: $B \leftarrow BU^{-1}$,$B \leftarrow L^{-1}B$

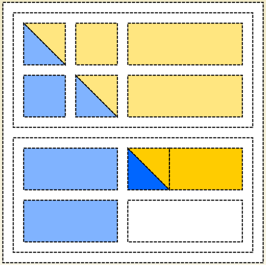
`gemm`: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative



Slab Recursive

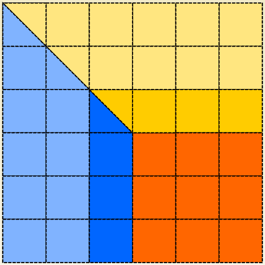


Tiled Recursive

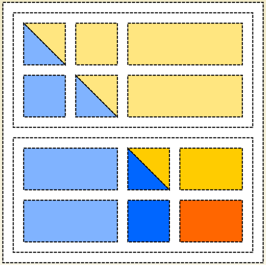
getrf: $A \to L, U$

Block algorithms

Tiled Iterative



Slab Recursive



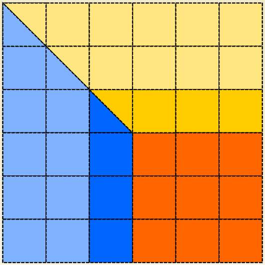
Tiled Recursive

trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

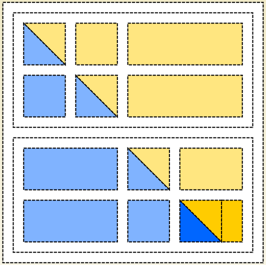
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative



Slab Recursive

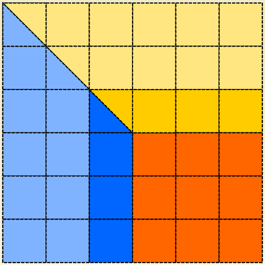


Tiled Recursive

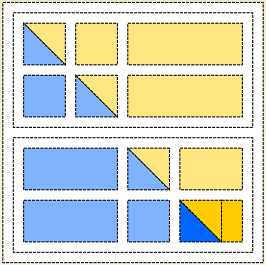
getrf: $A \rightarrow L, U$

Block algorithms

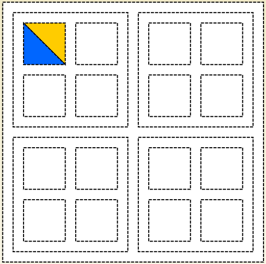
Tiled Iterative



Slab Recursive



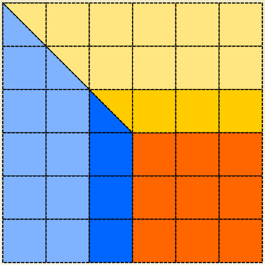
Tiled Recursive



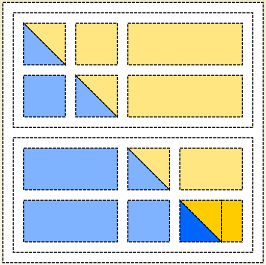
getrf: $A \to L, U$

Block algorithms

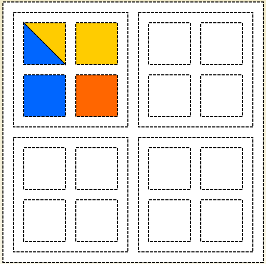
Tiled Iterative



Slab Recursive



Tiled Recursive

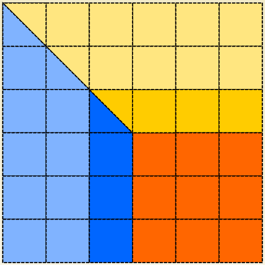


trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

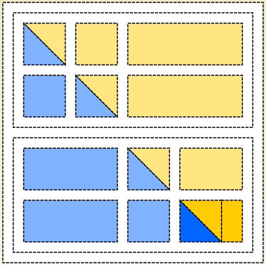
gemm: $C \leftarrow C - A \times B$

Block algorithms

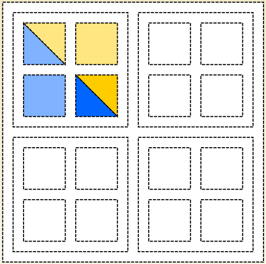
Tiled Iterative



Slab Recursive



Tiled Recursive



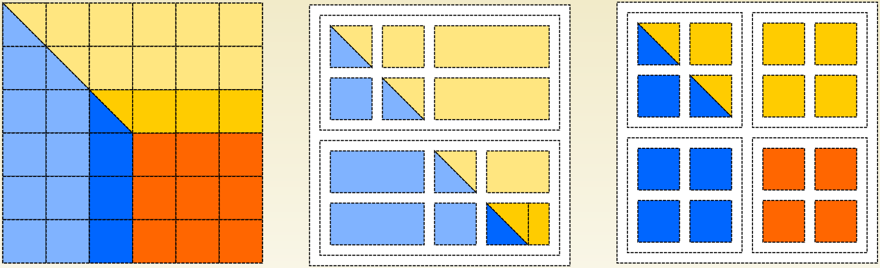
getrf: $A \rightarrow L, U$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive

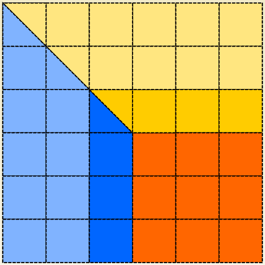


trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

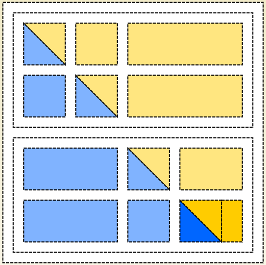
gemm: $C \leftarrow C - A \times B$

Block algorithms

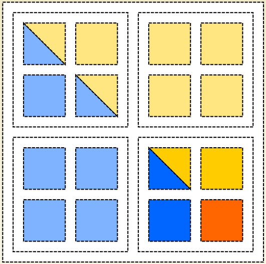
Tiled Iterative



Slab Recursive



Tiled Recursive



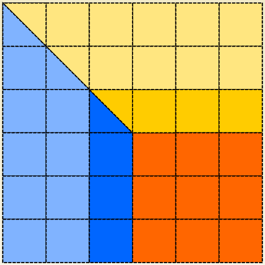
`getrf`: $A \to L, U$

`trsm`: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

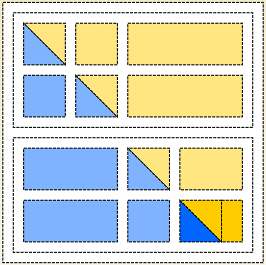
`gemm`: $C \leftarrow C - A \times B$

Block algorithms

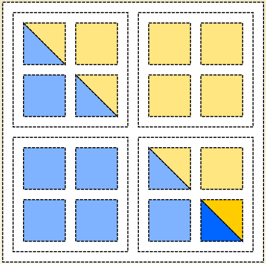
Tiled Iterative



Slab Recursive



Tiled Recursive



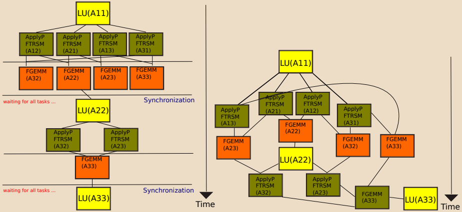
getrf: $A \rightarrow L, U$

Need for a high level parallel programming environment

Features required

Portability, Performance and Scalability. But more precisely:

- Runtime system with good performance for recursive tasks.
- Dataflow **task** synchronization



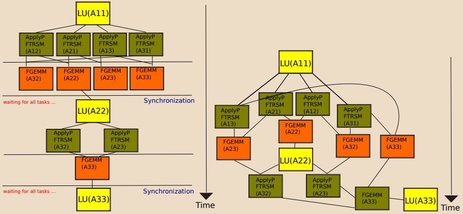
- Handle efficiently unbalanced workloads.
- Efficient range cutting for parallel for.

Need for a high level parallel programming environment

Features required

Portability, Performance and Scalability. But more precisely:

- Runtime system with good performance for recursive tasks.
- Dataflow **task** synchronization



- Handle efficiently unbalanced workloads.
- Efficient range cutting for parallel for.

→ Wish to design a code independently from the runtime system

→ Using runtime systems as a plugin

Outline

Runtime systems to be supported

OpenMP3.x and 4.0 supported directives: (using libgomp)

- Data sharing attributes:
  - OMP3 `shared`: data visible and accessible by all threads
  - OMP3 `firstprivate`: local copy of original value
  - OMP4 `depend`: set data dependencies
- Synchronization clauses: `#pragma omp taskwait`

xKaapi: via the libkomp [BDG12] library:

- OpenMP directives → xKaapi tasks.
- Re-implem. of task handling and management.
- Better recursive tasks execution.

TBB: designed for nested and recursive parallelism

- `parallel_for`
- `tbb::task_group, wait(), run()` using C++11 lambda functions

PALADIn

Parallel **A**lgebraic **L**inear **A**lgebra **D**edicated **In**terface

Mainly macro-based keywords

- No function call runtime overhead when using macros.
- No important modifications to be done to original program.
- Macros can be used also for C-based libraries.

Complementary C++ template functions

- Implement the different cutting strategies.
- Store the iterators

PALADIn description: task parallelism

Task parallelization: fork-join and dataflow models

- `PAR_BLOCK`: opens a parallel region.
- `SYNCH_GROUP`: Group of tasks synchronized upon exit.
- `TASK`: creates a task.
  - `REFERENCE(args...)`: specify variables captured by reference. By default all variables accessed by value.
  - `READ(args...)`: set var. that are read only.
  - `WRITE(args...)`: set var. that are written only.
  - `READWRITE(args...)`: set var. that are read then written.

Example:

```
void axpy(const Element a, const Element b, Element &y){y += a*x;}
SYNCH_GROUP(
            TASK(MODE(READ(a,x) READWRITE(y)),
                 axpy(a,x,y));
           );
```

Outline

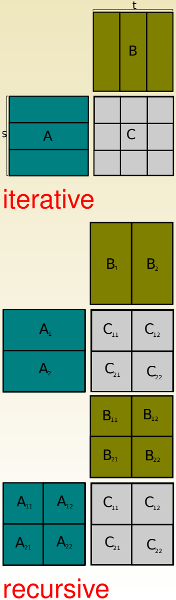
Parallel matrix multiplication

Iterative variants

- Fixed block size (`FIXED, GRAIN`)
  - Better control of data mapping in memory
  - Complexity: $O(n^3)$
- Fixed number of tasks (`THREADS`)
  - Less control of data mapping in memory
  - Complexity: $O(n^\omega)$

Recursive variants

- Almost no control of data mapping in memory
- Complexity: $O(n^\omega)$ or $O(n^3)$



Performance of pfgemm

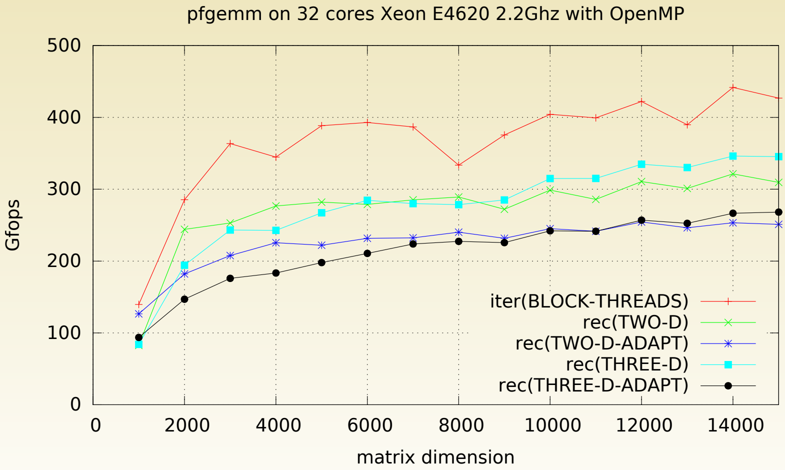


Figure: Speed of MatMul variants using OpenMP tasks

Performance of pfgemm

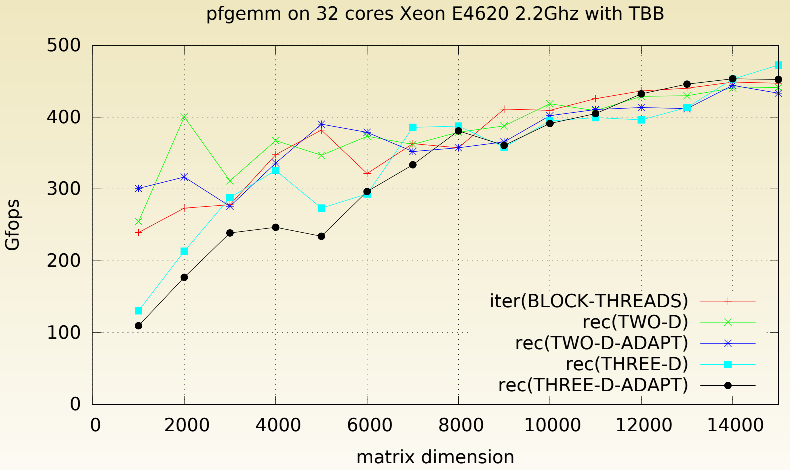


Figure: Speed of MatMul variants using IntelTBB tasks

Performance of pfgemm

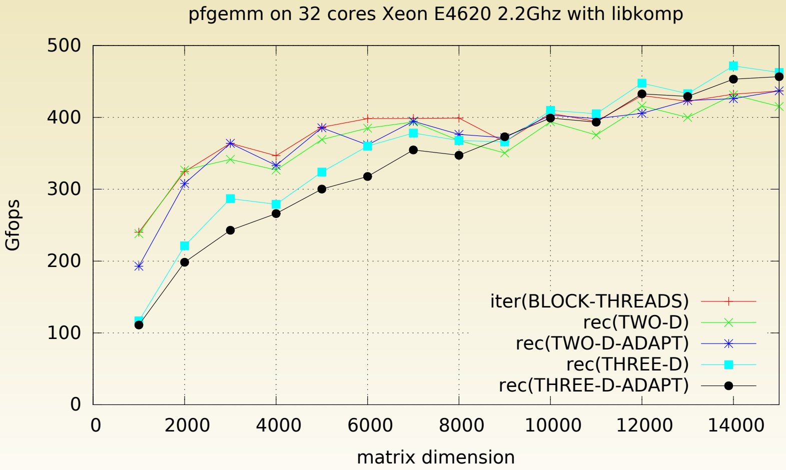
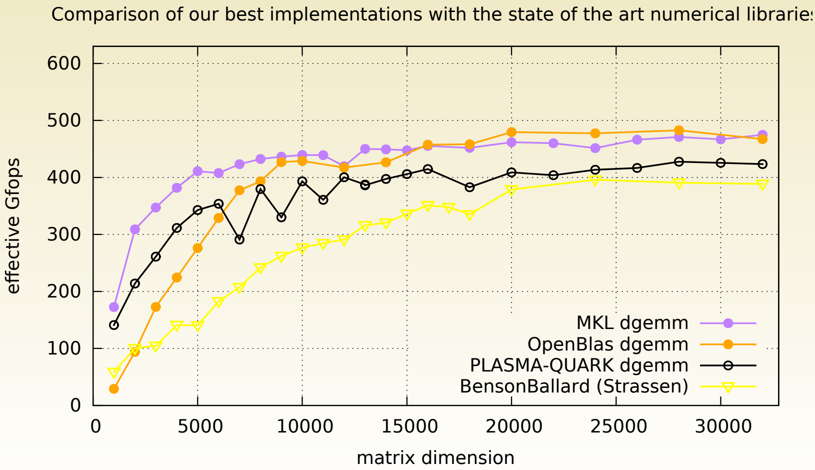


Figure: Speed of MatMul variants using XKaapi tasks

Parallel Matrix Multiplication: State of the art

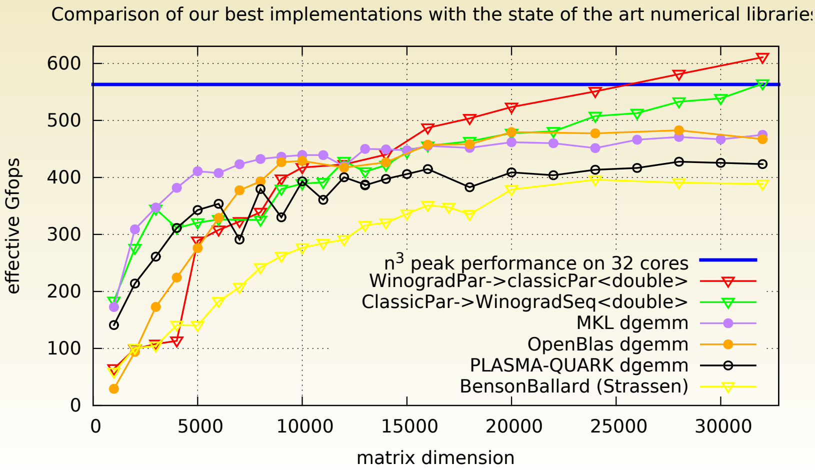
HPAC server: 32 cores Xeon E4620 2.2Ghz (4 NUMA sockets)



Parallel Matrix Multiplication: State of the art

HPAC server: 32 cores Xeon E4620 2.2Ghz (4 NUMA sockets)

$$\text{Effective Gfops} = \frac{\text{\# of field ops using classic matrix product}}{\text{time}}.$$



Outline

Parallel Triangular Solving Matrix

Iterative variant:

$$\left[\ X_1 \mid \ldots \mid X_k\ \right] \leftarrow L^{-1} \left[\ B_1 \mid \ldots \mid B_k\ \right].$$

- The computation of each $X_i \leftarrow L^{-1} B_i$ is independent
- k sequential tasks set as the number of available threads

Recursive variant:

1: Split $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} L_1 & \\ L_2 & L_3 \end{bmatrix}^{-1} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

2: $X_1 \leftarrow L_1^{-1} B_1$

3: $X_2 \leftarrow B_2 - L_2 X_1$ // Parallel MatMul

4: $X_2 \leftarrow L_3^{-1} B X_2$

Parallel Triangular Solving Matrix

Iterative variant:

$$[\ X_1 \mid \dots \mid X_k\ ] \leftarrow L^{-1} [\ B_1 \mid \dots \mid B_k\ ].$$

- The computation of each $X_i \leftarrow L^{-1} B_i$ is independent
- k sequential tasks set as the number of available threads

Recursive variant:

1: Split $\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} L_1 & \\ L_2 & L_3 \end{bmatrix}^{-1} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

2: $X_1 \leftarrow L_1^{-1} B_1$

3: $X_2 \leftarrow B_2 - L_2 X_1$ // Parallel MatMul

4: $X_2 \leftarrow L_3^{-1} B X_2$

Hybrid PFTRSM: column dimension of B small

- use iterative splitting in priority
- when $\#cols(X) < \#proc$: save some threads for recursive calls

Parallel Triangular Solving Matrix Experiments

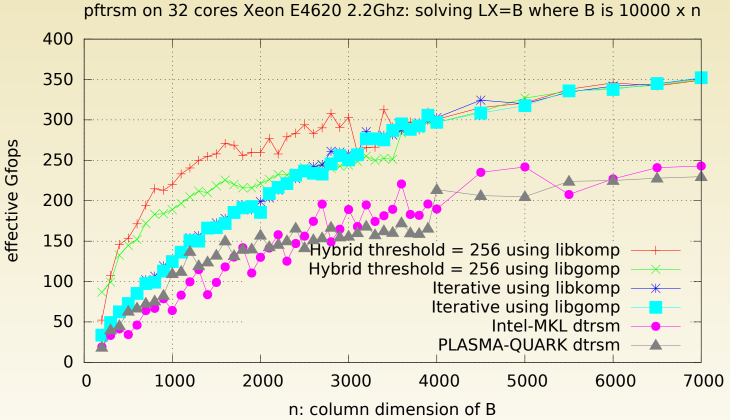


Figure: Comparing the Iterative and the Hybrid variants for parallel FTRSM using libkomp and libgomp. Only the outer dimension varies: B and X are $10000 \times n$.

Outline

Gaussian elimination design

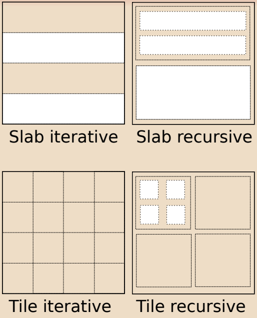
Reducing to MatMul: block versions

→ Asymptotically faster ($O(n^{\omega})$)
→ Better cache efficiency

Variants of block versions

Split on one dimension:
→ Row or Column slab cutting

Split on 2 dimensions:
→ Tile cutting



Gaussian elimination design

Reducing to MatMul: block versions

→ Asymptotically faster ($O(n^\omega)$)
→ Better cache efficiency

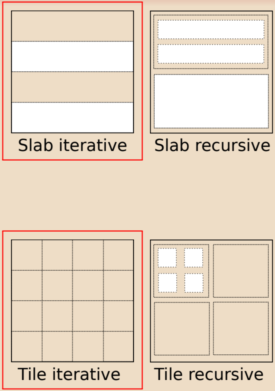
Variants of block versions

Iterative:

- Static → better data mapping control
- Dataflow parallel model → less sync

Recursive:

- Adaptive
- sub-cubic complexity
- No Dataflow → more sync



Gaussian elimination design

Reducing to MatMul: block versions

→ Asymptotically faster ($O(n^\omega)$)
→ Better cache efficiency

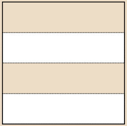
Variants of block versions

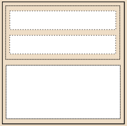
Iterative:

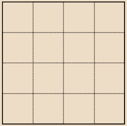
- Static → better data mapping control
- Dataflow parallel model → less sync

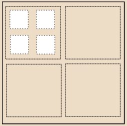
Recursive:

- Adaptive
- sub-cubic complexity
- No Dataflow → more sync

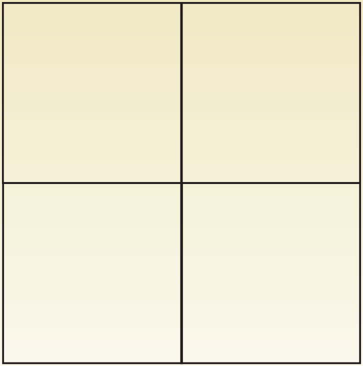

Slab iterative


Slab recursive


Tile iterative

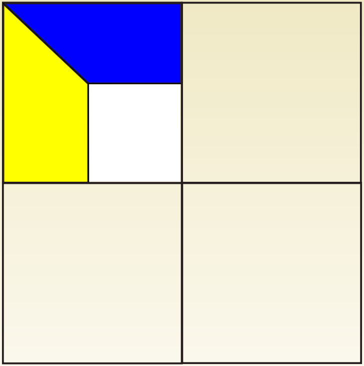

Tile recursive

Parallel tile recursive PLUQ algorithm



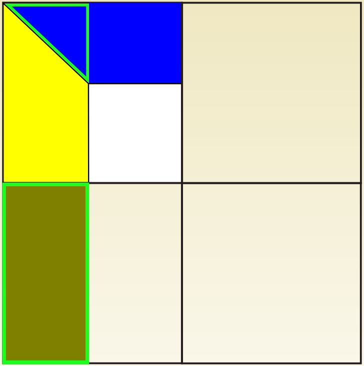
2×2 block splitting

Parallel tile recursive PLUQ algorithm



Recursive call

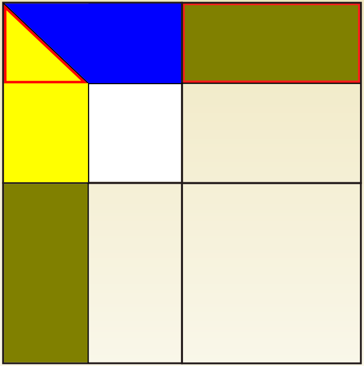
Parallel tile recursive PLUQ algorithm



pTRSM: $B \leftarrow BU^{-1}$

```
TASK(MODE(READ(A) READWRITE(B)),
     pftrsm(..., A, lda, B, ldb));
```

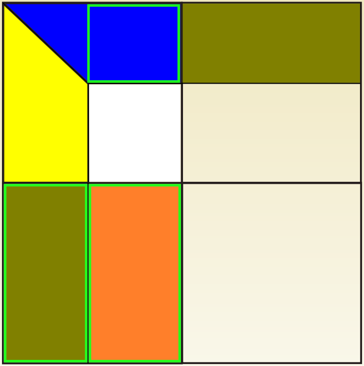
Parallel tile recursive PLUQ algorithm



pTRSM: $B \leftarrow L^{-1}B$

```
TASK(MODE(READ(A) READWRITE(B)),
     pftrsm(..., A, lda, B, ldb));
```

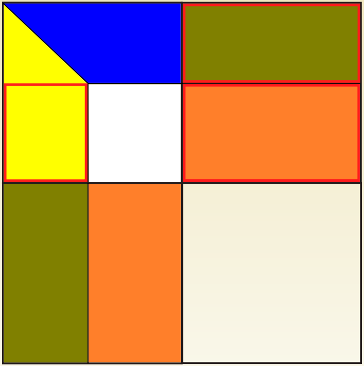
Parallel tile recursive PLUQ algorithm



pfgemm: $C \leftarrow C - A \times B$

```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

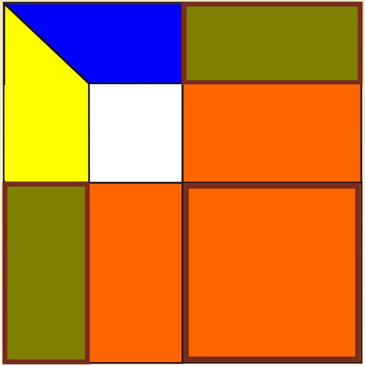
Parallel tile recursive PLUQ algorithm



pfgemm: $C \leftarrow C - A \times B$

```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

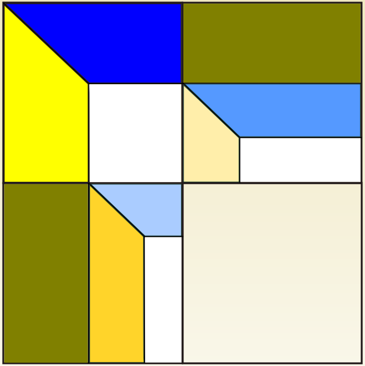
Parallel tile recursive PLUQ algorithm



`pfgemm`: $C \leftarrow C - A \times B$

```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

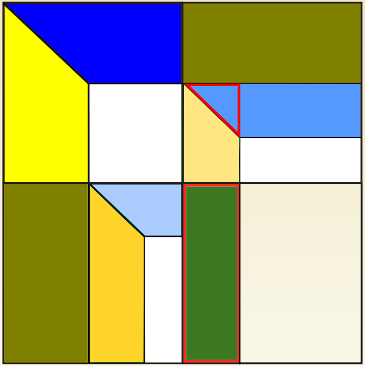
Parallel tile recursive PLUQ algorithm



2 independent recursive calls (concurrent → tasks)

```
TASK(MODE(READWRITE(A)),
     ppluq(..., A, lda));
```

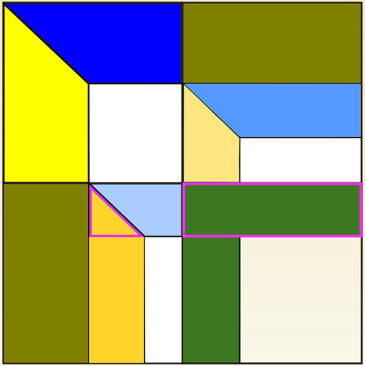
Parallel tile recursive PLUQ algorithm



pTRSM: $B \leftarrow BU^{-1}$

```
TASK(MODE(READ(A) READWRITE(B)),
     pftrsm(..., A, lda, B, ldb));
```

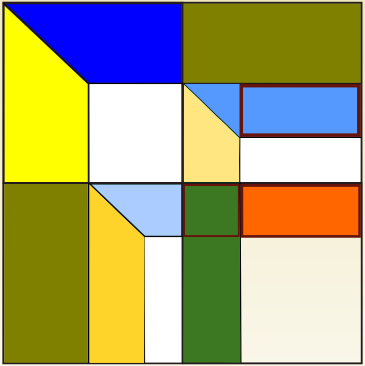
Parallel tile recursive PLUQ algorithm



pTRSM: $B \leftarrow L^{-1}B$

```
TASK(MODE(READ(A) READWRITE(B)),
     pftrsm(..., A, lda, B, ldb));
```

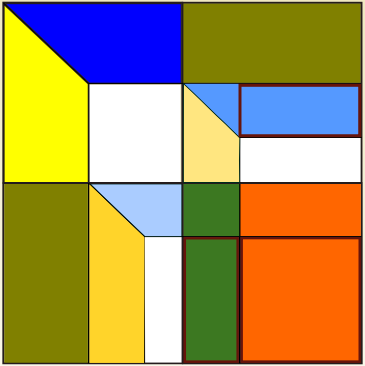
Parallel tile recursive PLUQ algorithm



pfgemm: $C \leftarrow C - A \times B$

```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

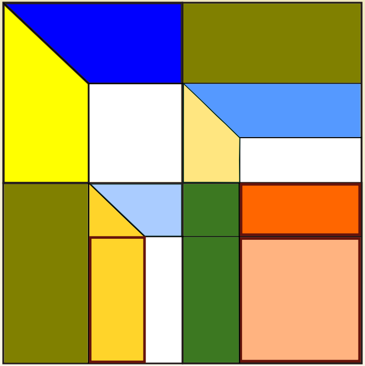
Parallel tile recursive PLUQ algorithm



pfgemm: $C \leftarrow C - A \times B$

```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

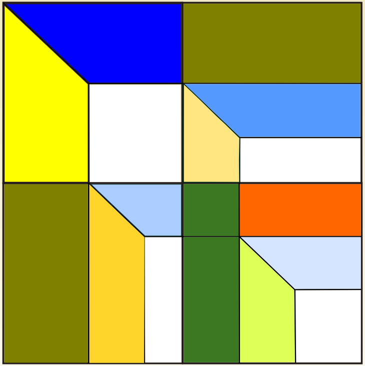
Parallel tile recursive PLUQ algorithm



pfgemm: $C \leftarrow C - A \times B$

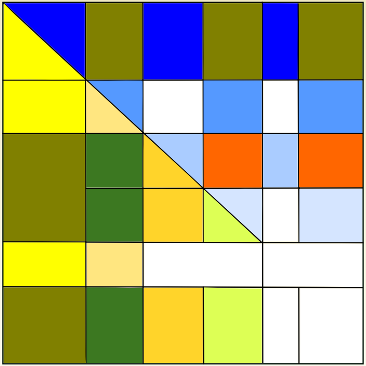
```
TASK(MODE(READ(A,B) READWRITE(C)),
     pfgemm(..., A, lda, B, ldb));
```

Parallel tile recursive PLUQ algorithm



Recursive call

Parallel tile recursive PLUQ algorithm



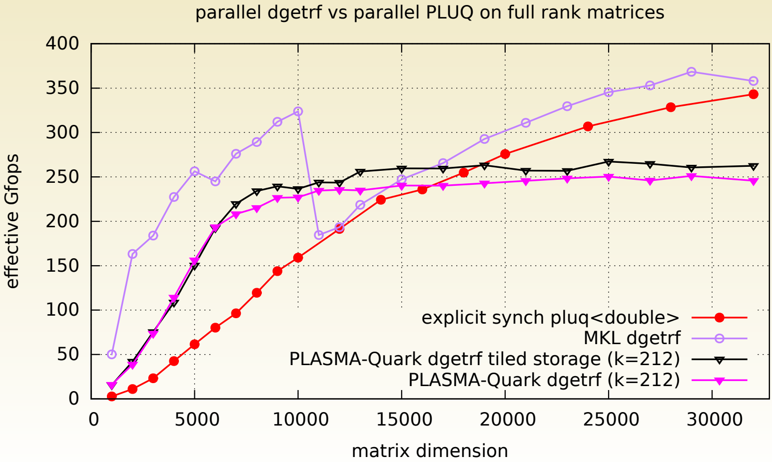
Puzzle game (block permutations)

Tile rec: better data locality and more square blocks for M.M.

State of the art: exact vs numerical linear algebra

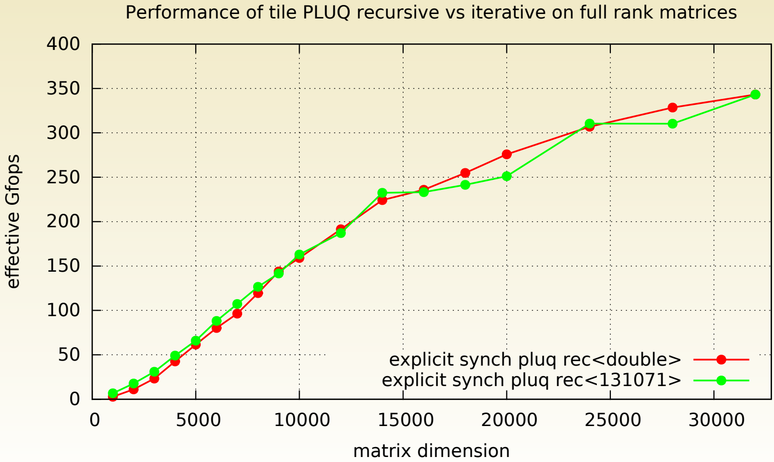
State of the art comparison:

- Exact PLUQ using PALADIn language: best performance with xKaapi
- Numerical LU (dgetrf) of PLASMA-Quark and MKL dgetrf

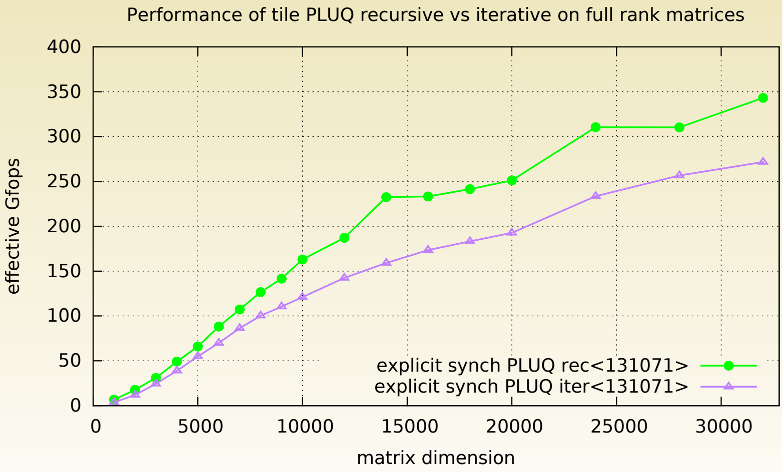


Performance of parallel PLUQ decomposition

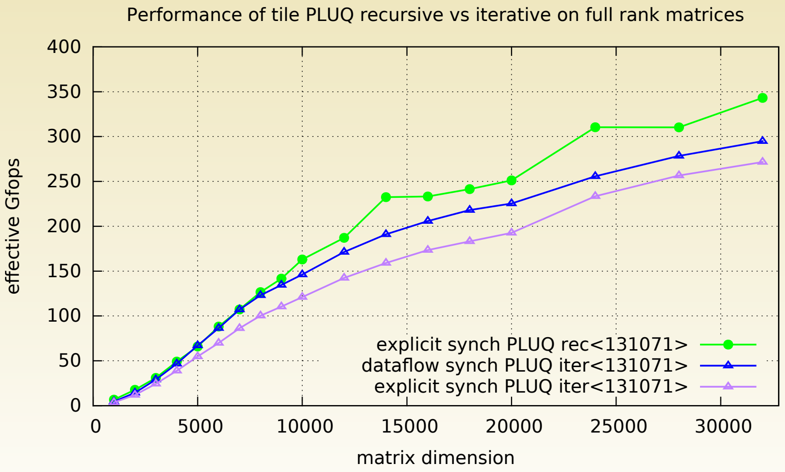
Low impact of modular reductions in parallel
→ Efficient SIMD implementation



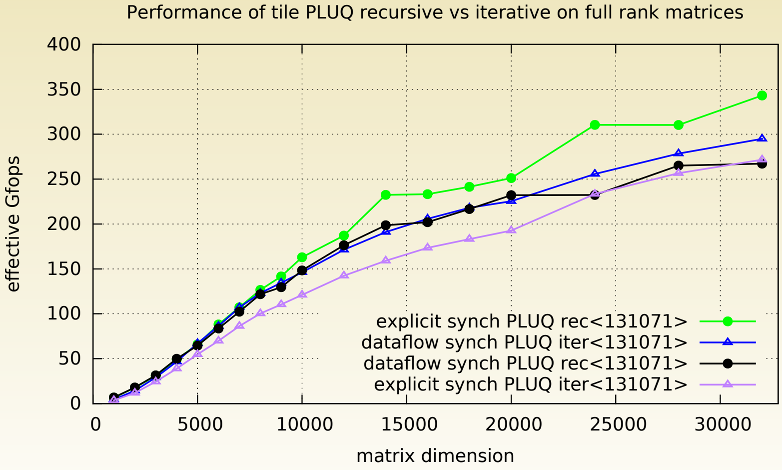
Performance of task parallelism: dataflow model



Performance of task parallelism: dataflow model



Performance of task parallelism: dataflow model



Possible improvement: implementation of the delegation of recursive tasks dependencies (Postpone access mode in the parallel programming environments)

Outline

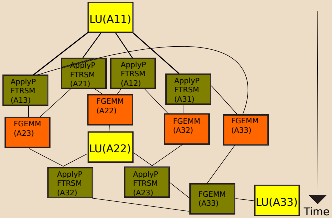
Conclusion

Lessons learnt for the parallelization of LU over $\mathbb{Z}/p\mathbb{Z}$

- Blocking impacts arithmetic cost $\Rightarrow$fine granularity hurts
- Rank deficiency can offer more parallelism (cf. separators)
- sub-cubic perfs in parallel
- requires a runtime efficient for recursive tasks (XKaapi)

Perspectives

Data flow task dependencies



- already at use in tiled iterative algorithms (XKaapi)
- new challenges for recursive tasks:
  - Recursive inclusion of sub-matrices
  - Postponed modes (removing fake dependencies)
- Distributed on small sized clusters

Thank you