Look-Ahead in Dense Matrix Factorizations

Sandra Catalán, José R. Herrero, Enrique S. Quintana-Ortí,
Rafael Rodríguez-Sánchez, Robert van de Geijn

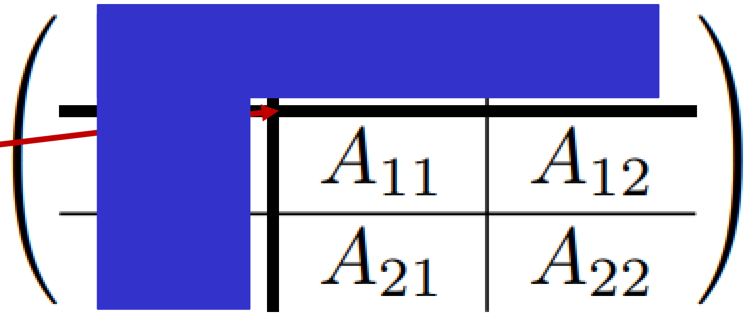
UNIVERSITAT
JAUME·I

UPC

THE UNIVERSITY OF TEXAS AT AUSTIN

The LU factorization (right-looking variant)

```
for ( k = 0; k < n / b; k+=b ) {




}
```



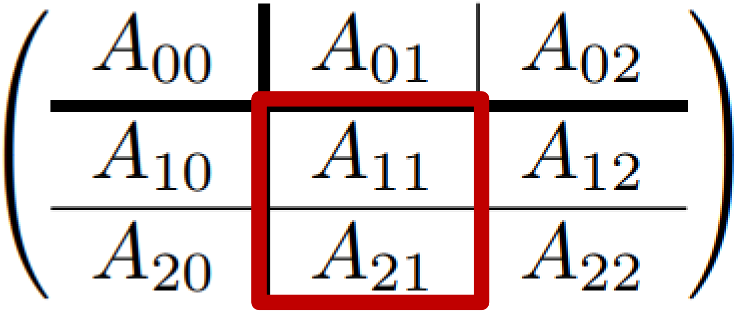
The LU factorization (right-looking variant)

```
for ( k = 0; k < n / b; k+=b ) {




}
```

$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

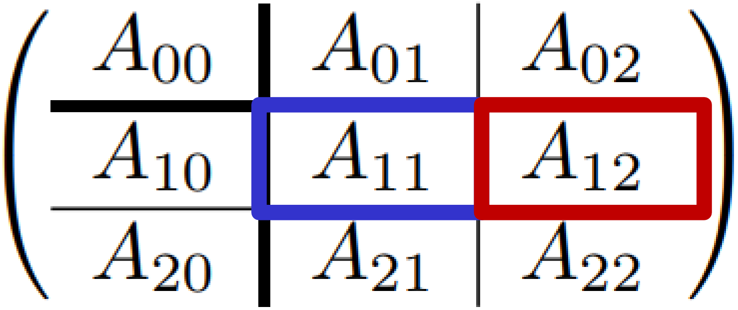
Block size
- Width of A_{11}
- Small to cast most computations on terms of efficient kernels

The LU factorization (right-looking variant)



```
for ( k = 0; k < n / b; k++ ) {
   getf2( &A(k,k) );







}
```

The LU factorization (right-looking variant)

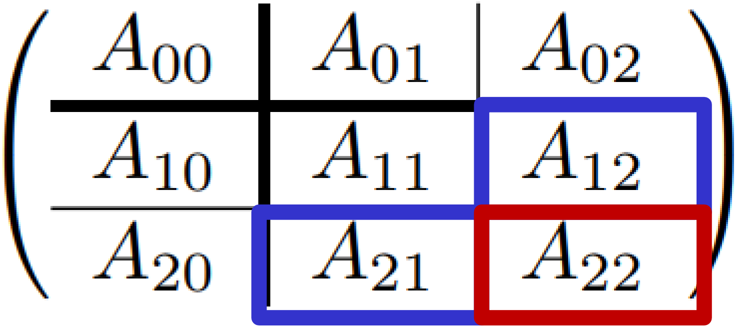


$$\left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & \boxed{A_{11}} & \boxed{A_{12}} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

```
for ( k = 0; k < n / b; k++ ) {

    getf2( &A(k,k) );

    trsm(  &A(k,k) ,  &A(k,k+b) );

}
```

Dependency: RL1 → RL2

The LU factorization (right-looking variant)



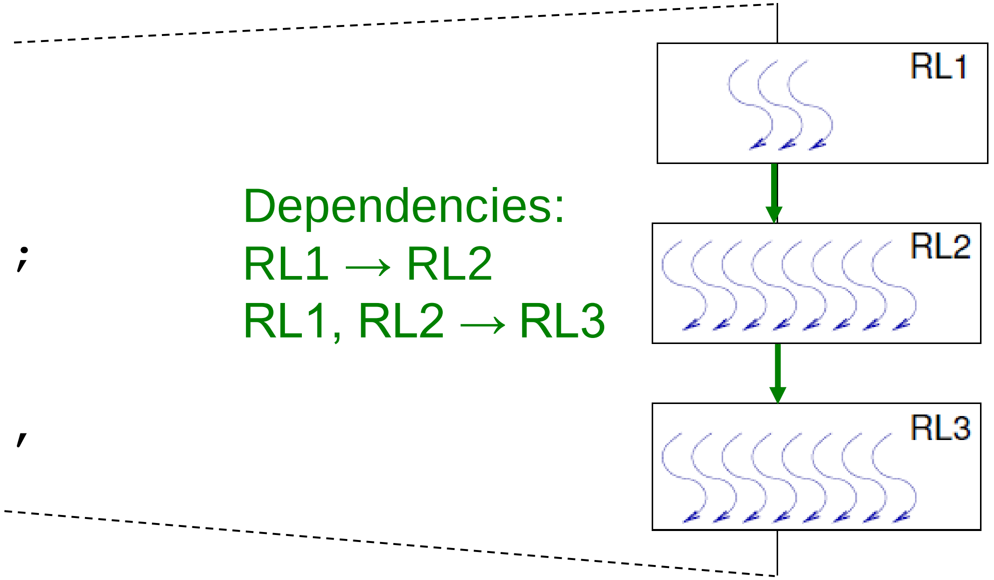
```
for ( k = 0; k < n / b; k++ ) {

    getf2( &A(k,k) );


    trsm(  &A(k,k),    &A(k,k+b) );

    gemm(  &A(k+b,k) , &A(k,k+b) , &A(k+b,k+b) );

}
```

Dependencies: RL1, RL2 → RL3

The LU factorization (right-looking variant)

Conventional parallelization:
Calls to multi-threaded BLAS

```
for ( k = 0; k < n / b; k++ ) {

    getf2( &A(k,k) );


    trsm(  &A(k,k),    &A(k,k+b) );


    gemm(  &A(k+b,k) , &A(k,k+b) ,
           &A(k+b,k+b) );

}
```



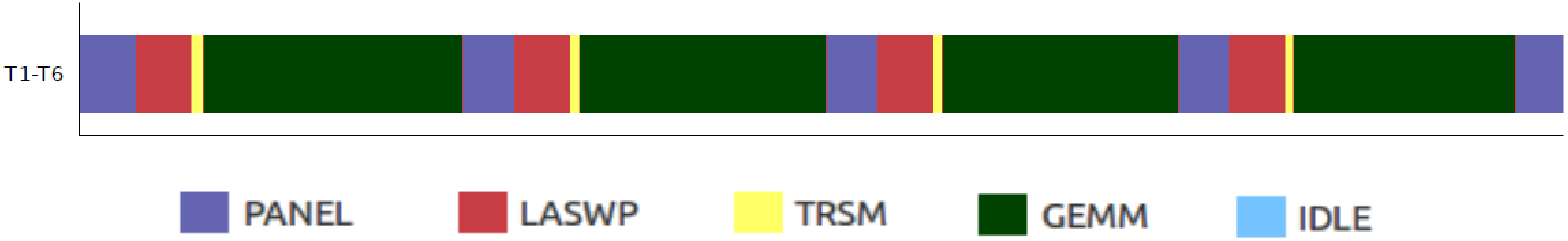
Dependencies:
RL1 → RL2
RL1, RL2 → RL3

The LU factorization

Intel Xeon E5-2603 v3 (Haswell, 6 cores)

- 10,000x10,000 matrix
- RL variant with $b=b_o=256$
- Calls to BLIS kernels for GEMM, TRSM
- Sequential LASWP
- Partial pivoting
- Call to GETRF, with $b_i=32$

→ 2% of flops in Panel Factorization (PF)



The LU (and other) factorization(s)

Avoiding the curse of PF:

T1) Exploit fine-grained parallelism within the panel (parallelization by rows)

- Usually limited parallelism

T2) Exploit intra-iteration parallelism: Decompose PF and update into multiple operations (algorithm-by-tiles or tile algorithms)

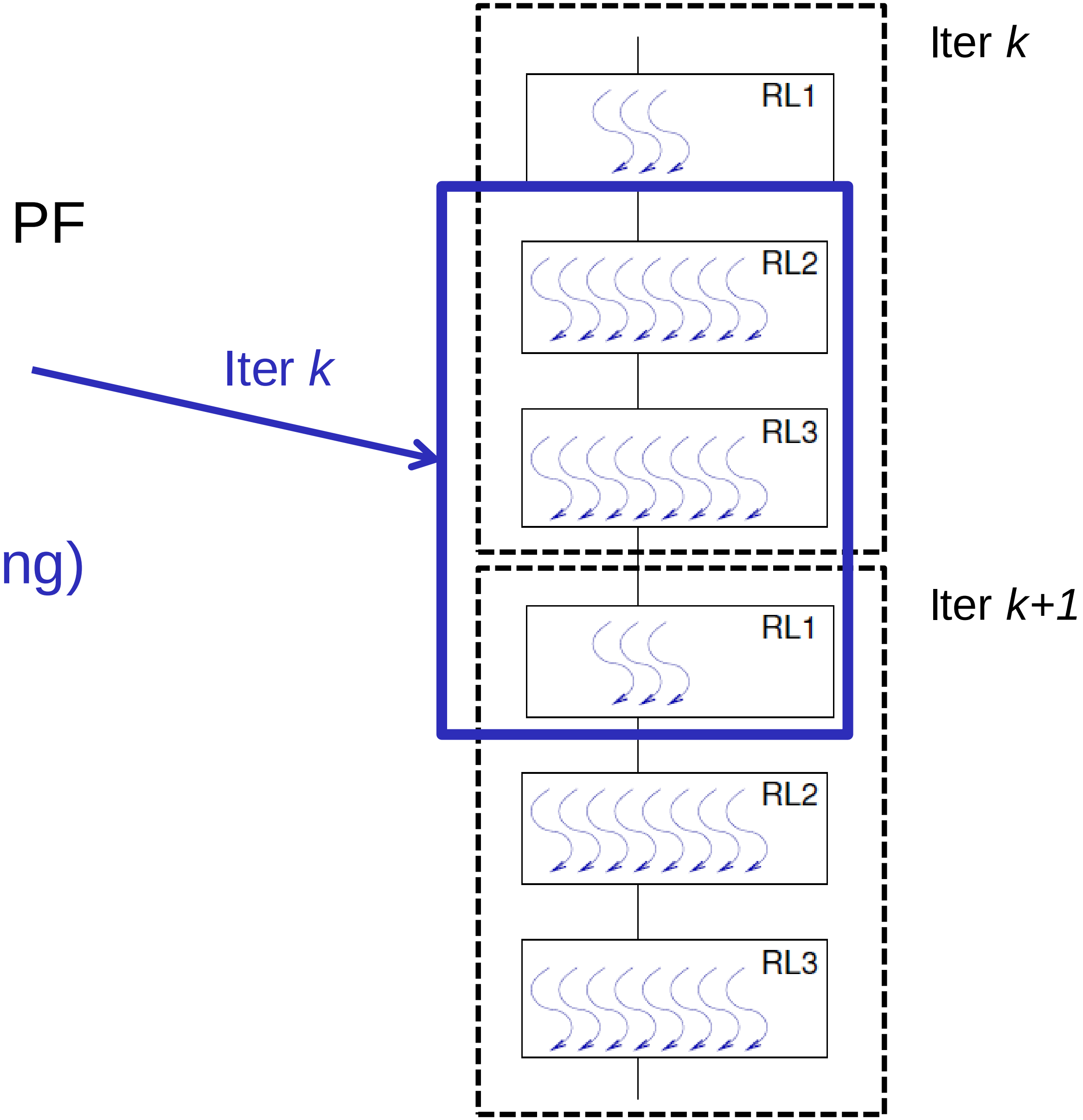
- Not always possible without changing the numerics (LU)
- In general, introduces overhead: more flops, repeated packing/unpacking in calls to small BLAS
- Runtime-assisted (cache-oblivious)
- Requires kernels that are rarely efficient on GPUs, or the “reconstruction” of the panel factorization

The LU (and other) factorization(s)

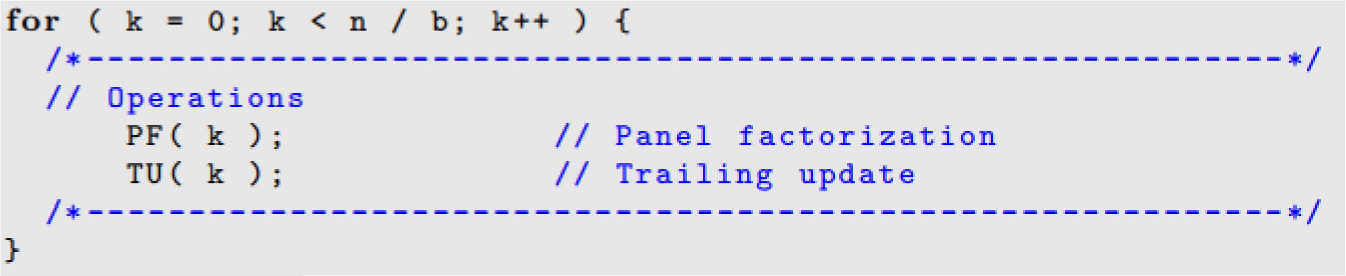
Avoiding the curse of PF:

T3) Exploit inter-iteration parallelism by overlapping PF with trailing update, also known as look-ahead!

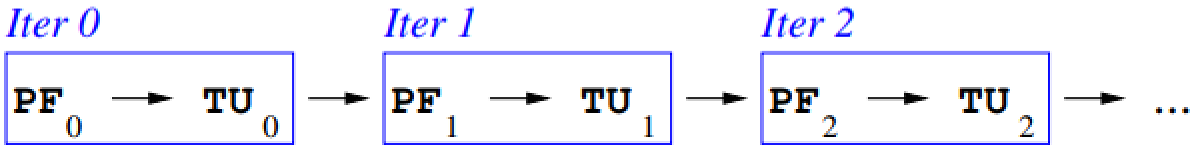
(similar to software pipelining)



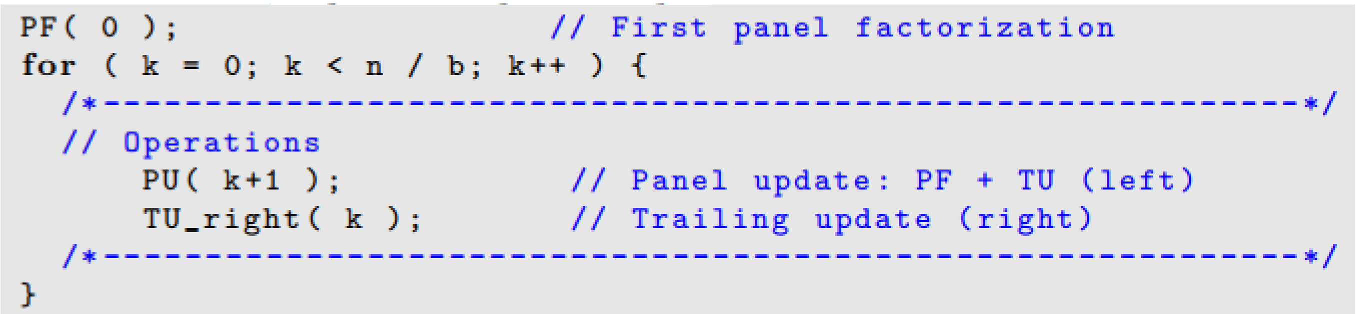
The LU factorization



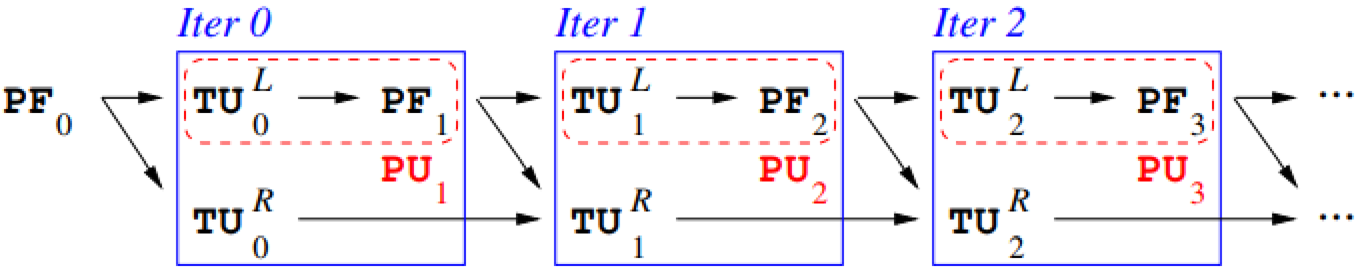
```
for ( k = 0; k < n / b; k++ ) {
  /*-----------------------------------------------------------*/
  // Operations
      PF( k );                 // Panel factorization
      TU( k );                 // Trailing update
  /*-----------------------------------------------------------*/
}
```



Loo-ahead: $\mathrm{TU}_k \rightarrow (\mathrm{TU}_k^L \mid \mathrm{TU}_k^R)$



```
PF( 0 );                      // First panel factorization
for ( k = 0; k < n / b; k++ ) {
  /*-----------------------------------------------------------*/
  // Operations
      PU( k+1 );                 // Panel update: PF + TU (left)
      TU_right( k );             // Trailing update (right)
  /*-----------------------------------------------------------*/
}
```



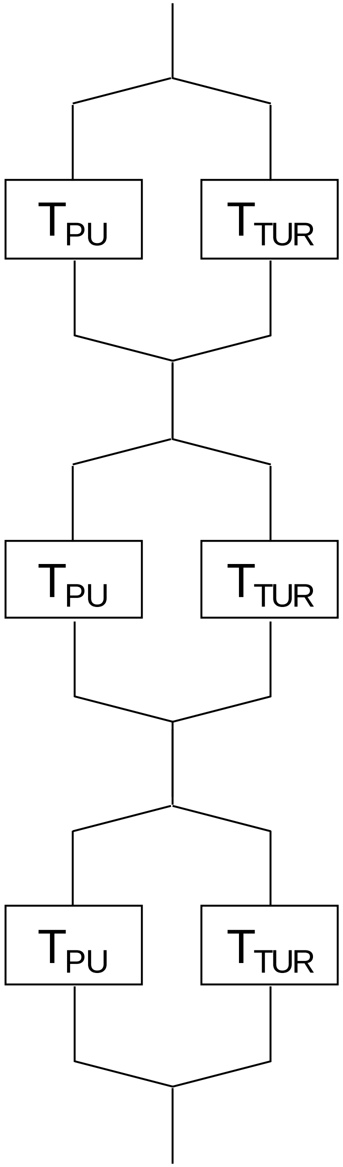
The LU (and other) factorization(s)

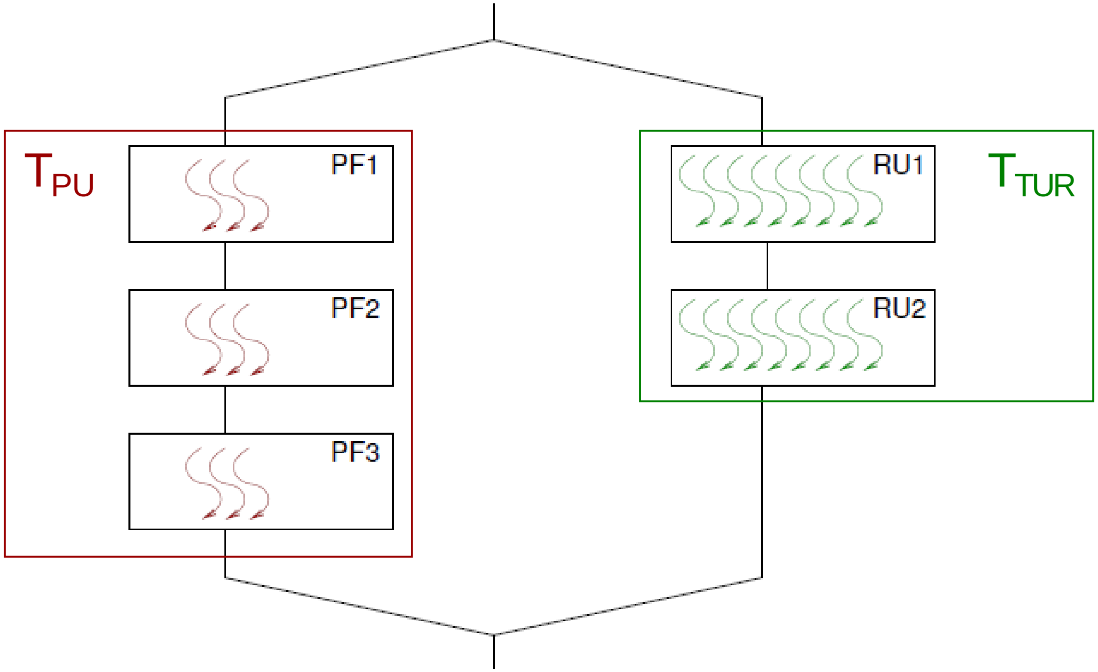
Look-ahead confused with T2 + runtime because the latter may yield the same effect (exploitation inter-iteration parallelism) transparently to the user

Not always (to be seen later)

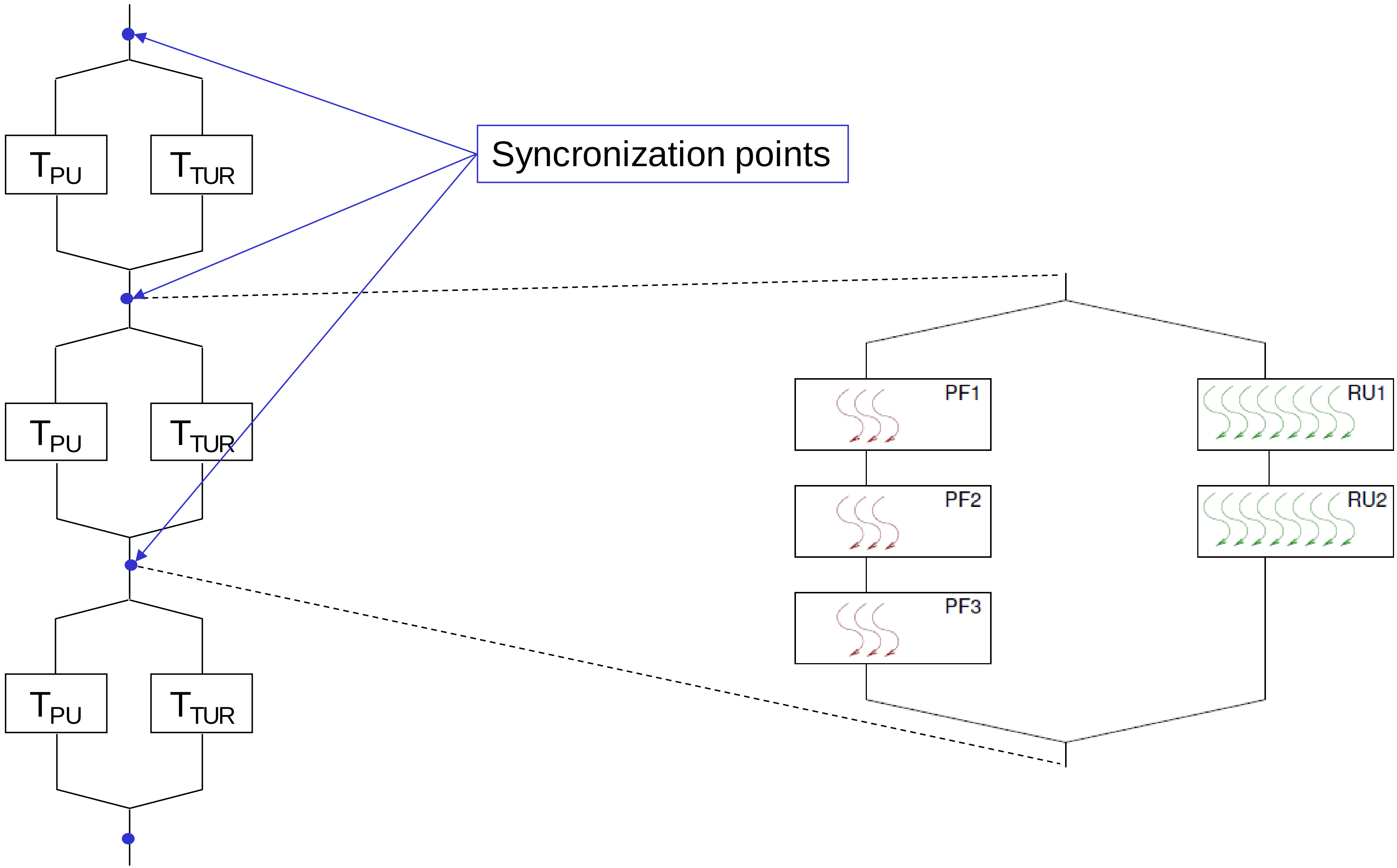
- Only look-ahead (potentially) eliminates PF from the algorithm's critical path
- Dynamic look-ahead forces threads to compete for shared resources (cache levels)

The LU factorization

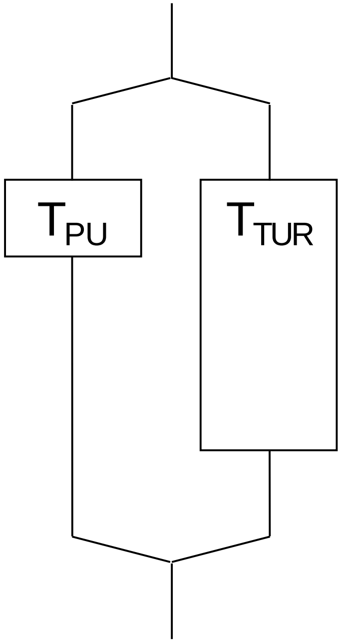




The LU factorization: What if $T_{PU} > T_{TUR}$ or vice-versa?

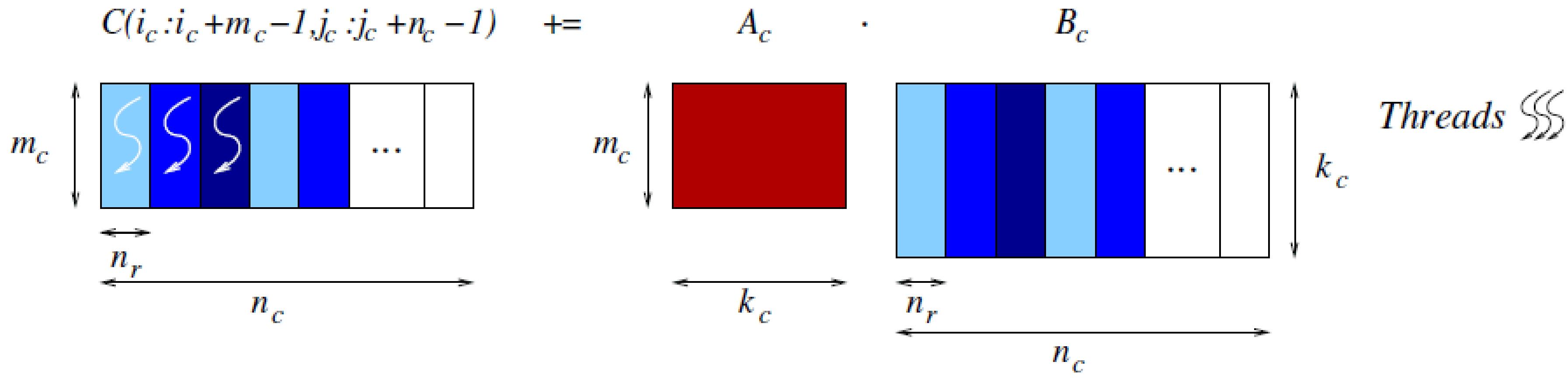


The LU factorization: $T_{TUR} > T_{PU}$

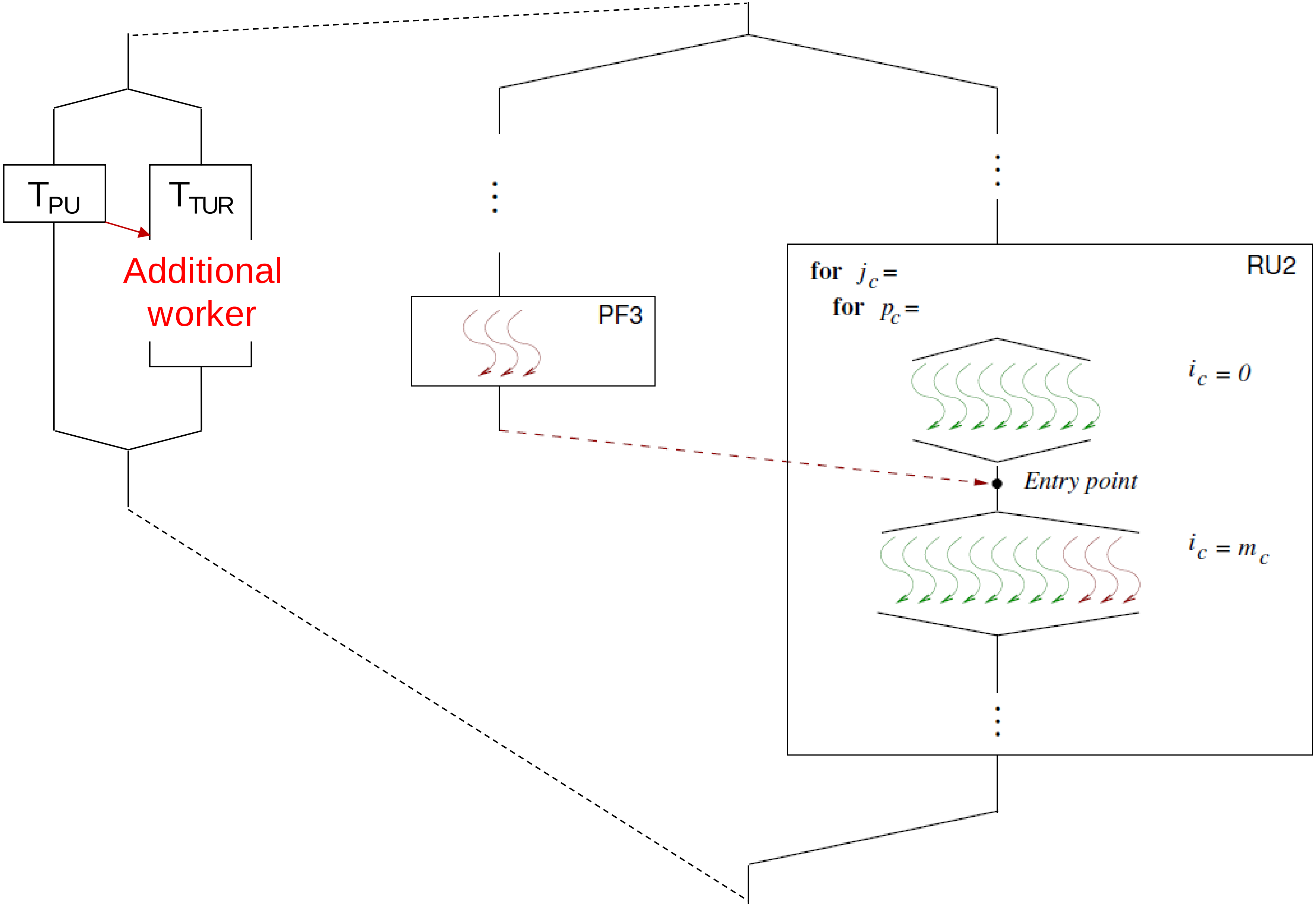


The LU factorization: $T_{TUR} > T_{PU}$. Malleable BLIS

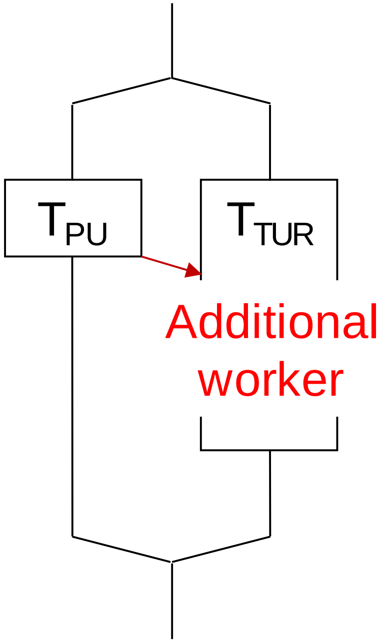
```
Loop 1  for $j_c = 0, \dots, n-1$ in steps of $n_c$
Loop 2    for $p_c = 0, \dots, k-1$ in steps of $k_c$
            $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$          // Pack into $B_c$
Loop 3      for $i_c = 0, \dots, m-1$ in steps of $m_c$
              $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$        // Pack into $A_c$
------------------------------------------------------------------------------------------
Loop 4        for $j_r = 0, \dots, n_c - 1$ in steps of $n_r$                      // Macro-kernel
Loop 5          for $i_r = 0, \dots, m_c - 1$ in steps of $m_r$
          ----------------------------------------------------------------------
                  $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$       // Micro-kernel
                      $+= \; A_c(i_r : i_r + m_r - 1, 0 : k_c - 1)$
                      $\cdot \; B_c(0 : k_c - 1, j_r : j_r + n_r - 1)$
          ----------------------------------------------------------------------
                endfor
              endfor
------------------------------------------------------------------------------------------
            endfor
          endfor
        endfor
```



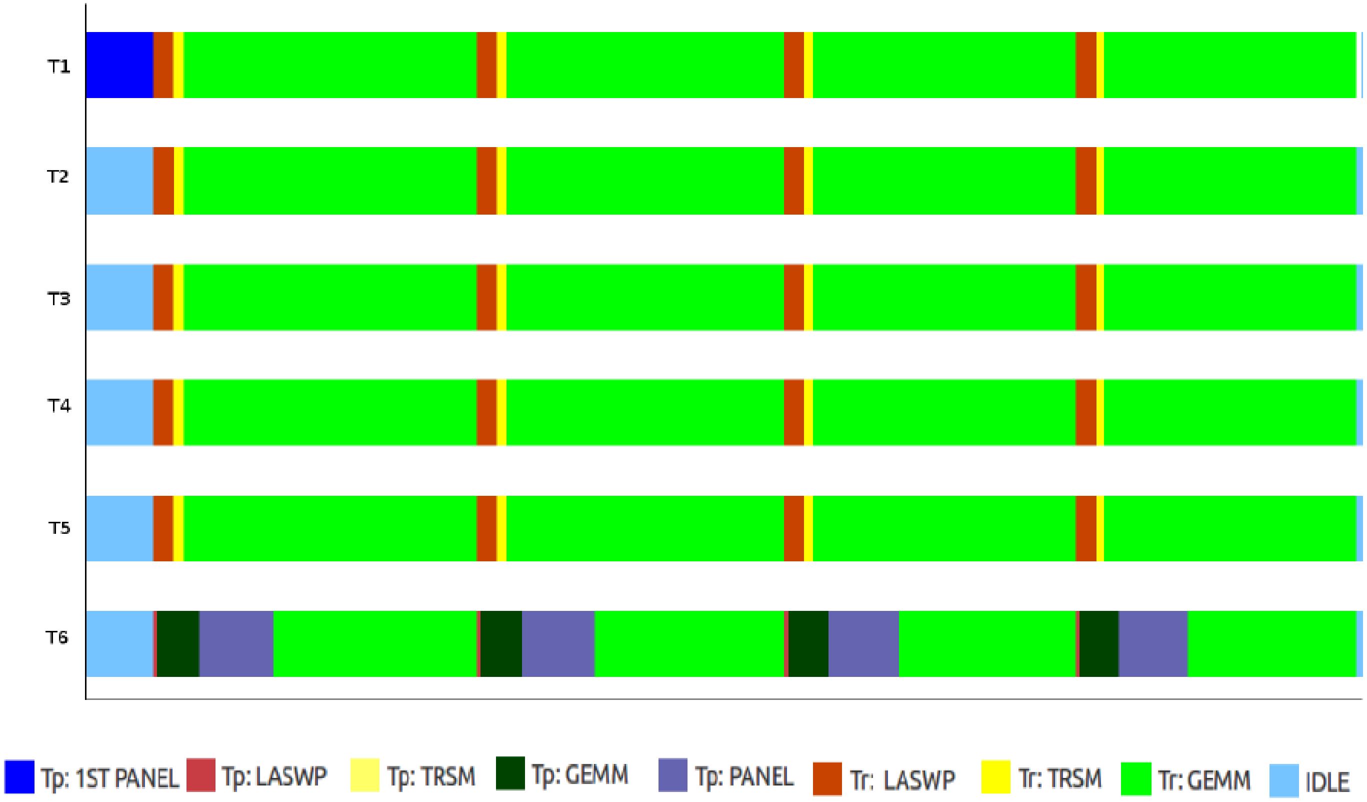
The LU factorization: $T_{TUR} > T_{PU}$. Malleable BLIS



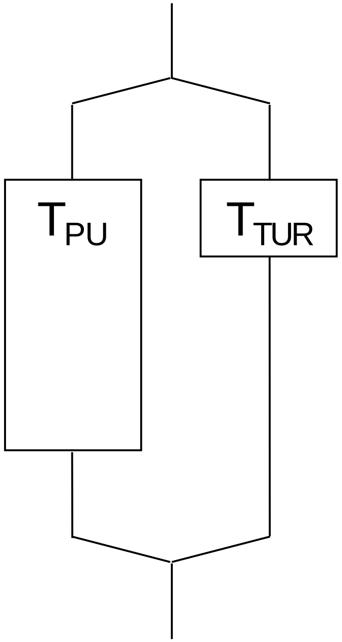
The LU factorization: $T_{TUR} > T_{PU}$. Malleable BLIS

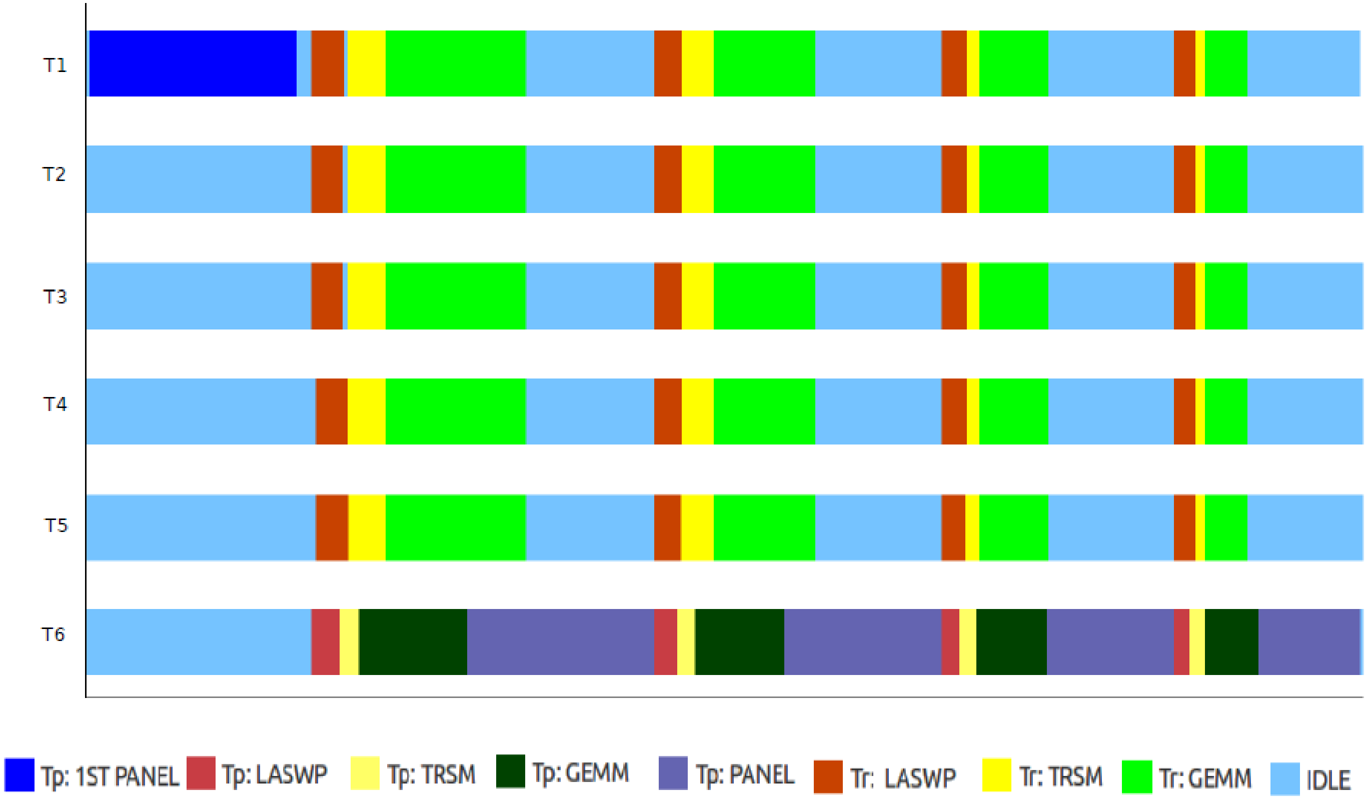


Additional worker

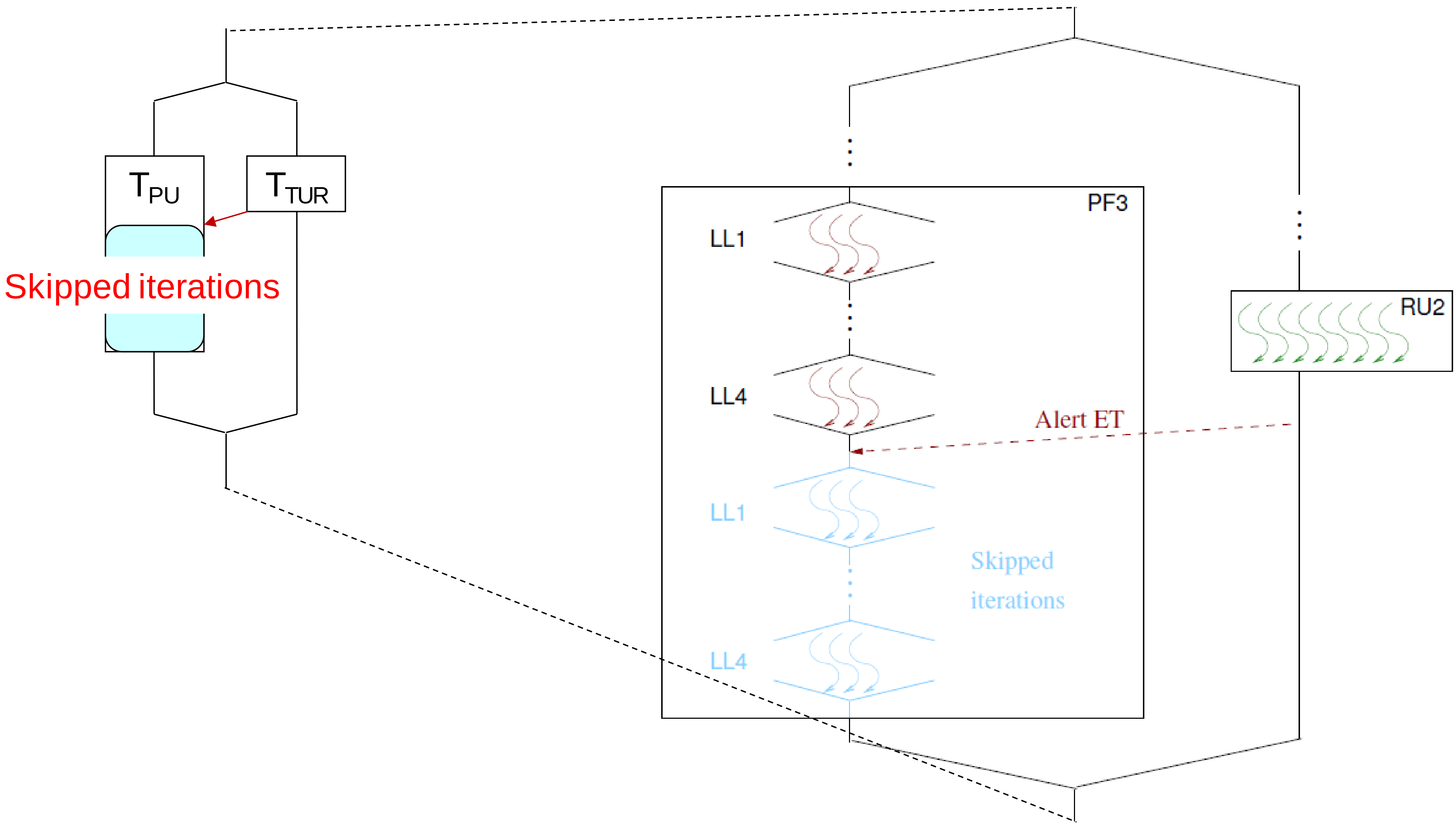


The LU factorization: $T_{PU} > T_{TUR}$

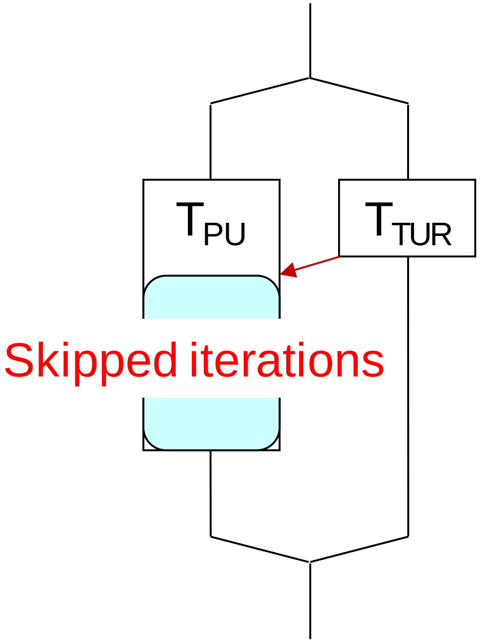




The LU factorization: $T_{PU} > T_{TUR}$. Early Termination (ET)



The LU factorization: $T_{PU} > T_{TUR}$. Early Termination (ET)



Automatic adaptive block size
RL vs Left-Looking (LL) variants:

RL1. $\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} := \text{LU\_UNB}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}\right)$

RL2. $A_{12} := \text{TRILU}(A_{11})^{-1} A_{12}$

RL3. $A_{22} := A_{22} - A_{21} A_{12}$

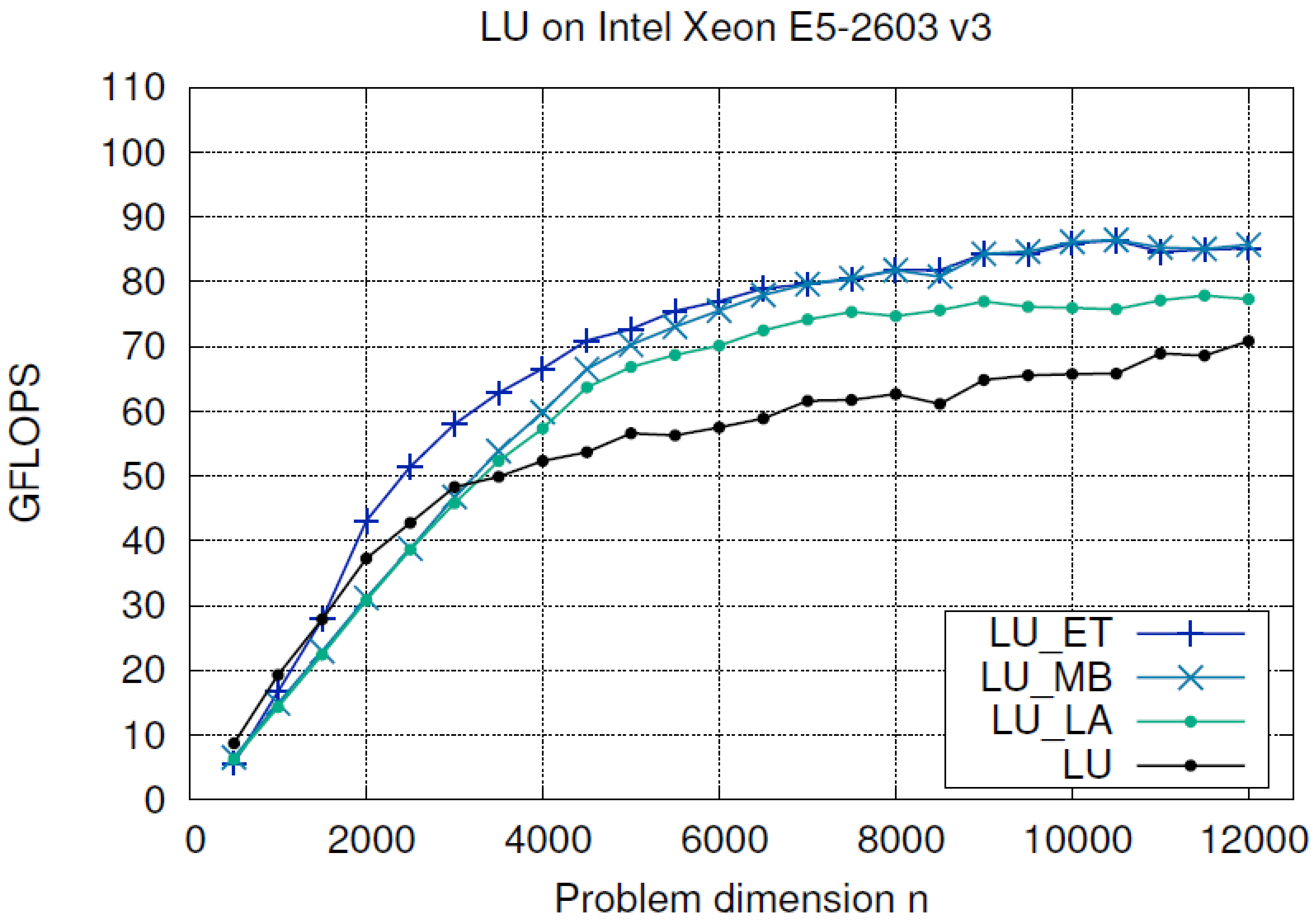
LL1. $A_{01} := \text{TRILU}(A_{00})^{-1} A_{01}$

LL2. $\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} := \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} - \begin{bmatrix} A_{10} \\ A_{20} \end{bmatrix} A_{01}$

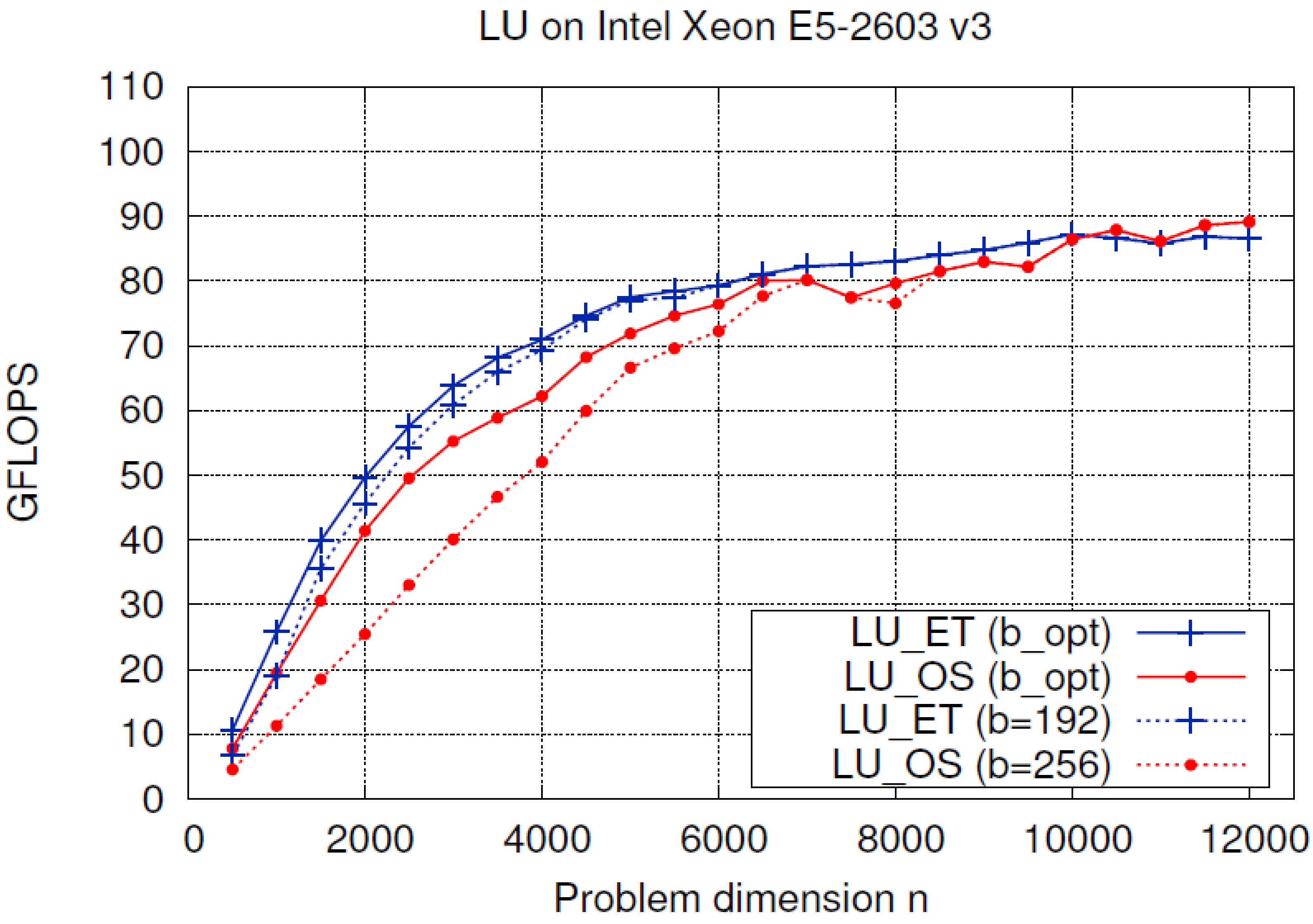
LL3. $\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} := \text{LU\_UNB}\left(\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}\right)$

LL delays computation to the end and, therefore, allows larger block sizes

The LU factorization: Experimental evaluation



The LU factorization: Experimental evaluation



The LU factorization: Summary

- Static look-ahead can be competitive with runtime-based approach
  - More cache-friendly than algorithms-by-blocks+runtime

- Same overhead, kernels and efficiency as standard right-looking algorithm
- Preserves the numerics (LU)
- ET automatically adjusts the block size

Other matrix factorizations

The PF "bottleneck" appears in several DLA operations:

- LU factorization
- QR factorization: Extension of look-ahead is trivial
- (To a minor extent) Cholesky factorization

- Two-sided factorizations:
  - Reduction from symmetric dense to band (SEVP)
  - Reduction from dense to triangular-band (SVD)

Look-ahead?

Reduction to triangular-band for SVD

- Upper bandwidth *w*
- Algorithmic block size *b* (for simplicity, *w* = *b*)
- At iteration *k*

1. LEFT PANEL FACTORIZATION:

$$B = Q_L R,$$

2. LEFT TRAILING UPDATE:

$$E := Q_L^T E = (I_i + W_L Y_L^T)^T E = E + Y_L (W_L^T E),$$

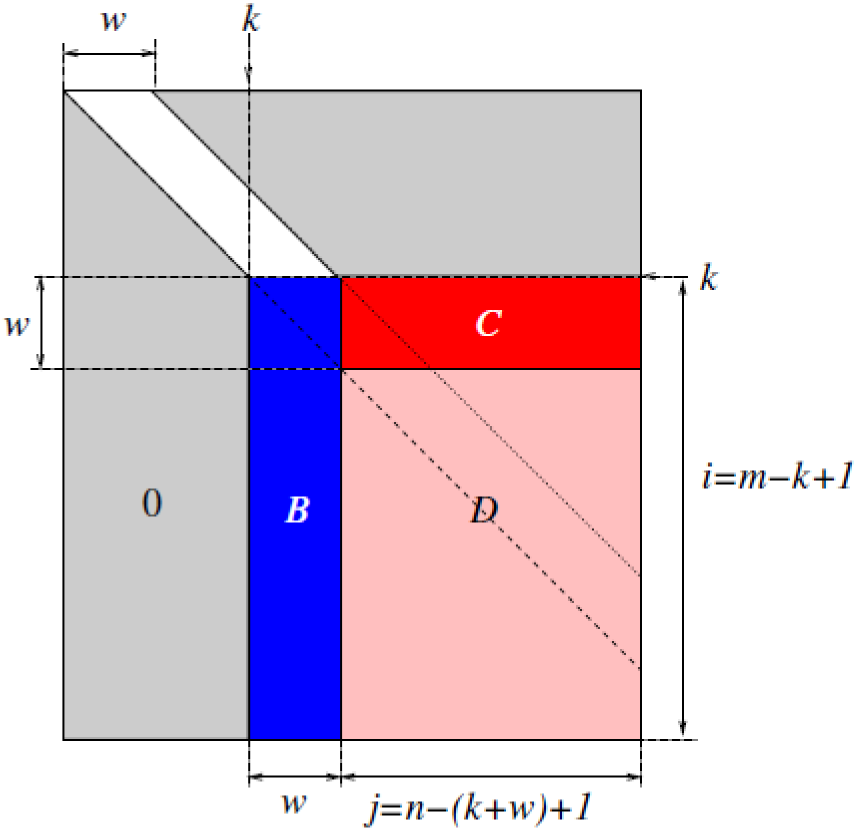
with $E = \begin{bmatrix} C \\ D \end{bmatrix} \in \mathbb{R}^{i \times j}$

3. RIGHT PANEL FACTORIZATION:

$$C = L Q_R^T,$$

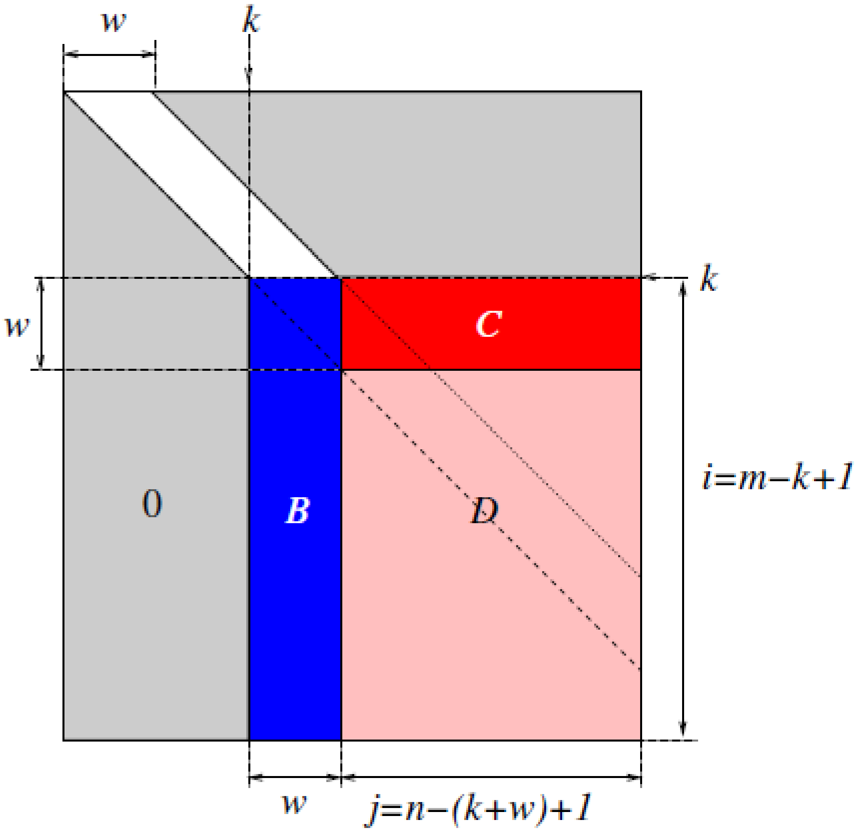
4. RIGHT TRAILING UPDATE:

$$D := D Q_R = D(I_j + W_R Y_R^T) = D + (D W_R) Y_R^T.$$



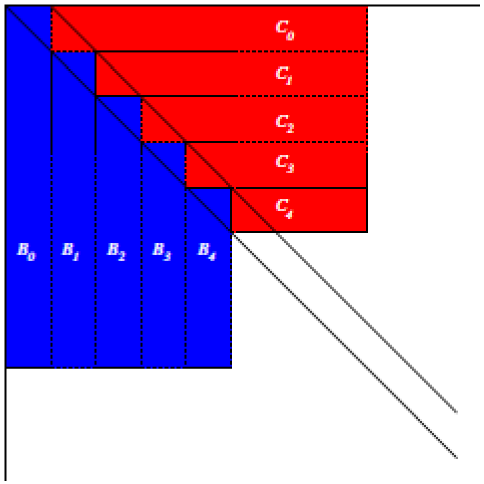
Reduction to triangular-band for SVD

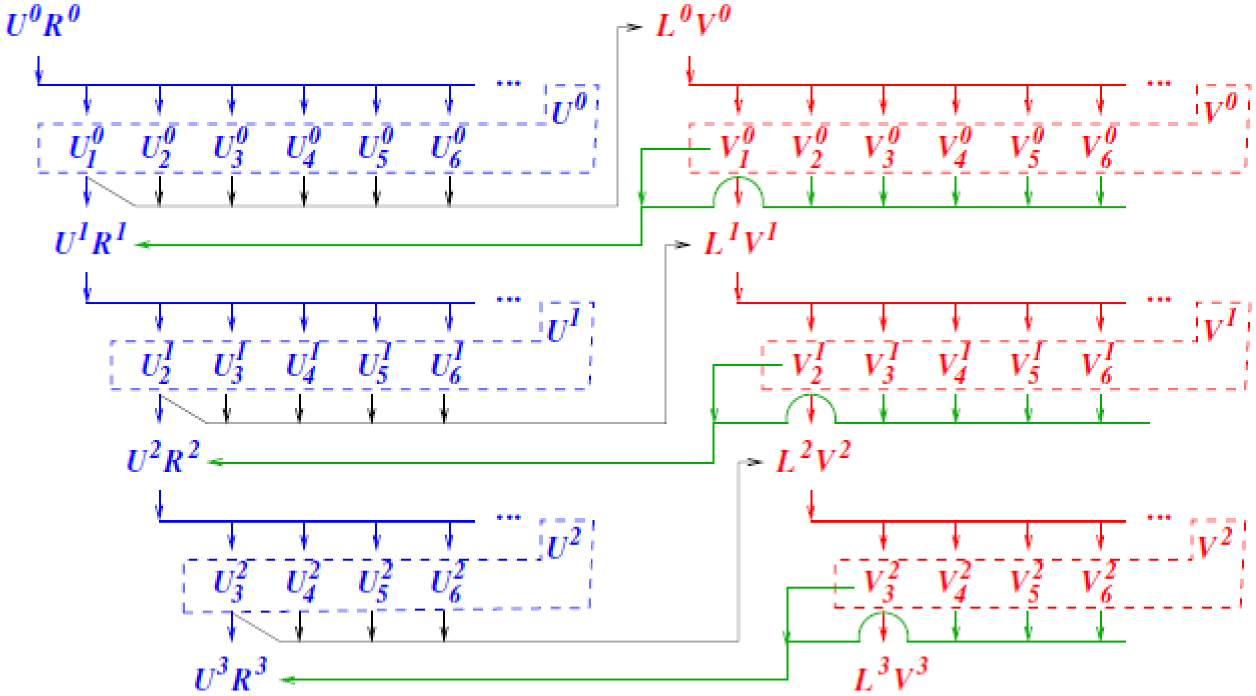
- For look-ahead, during iteration *k*:
  - Update current trailing submatrices w.r.t. current PF
  - Compute next PF



Reduction to triangular-band for SVD

- *w=b*

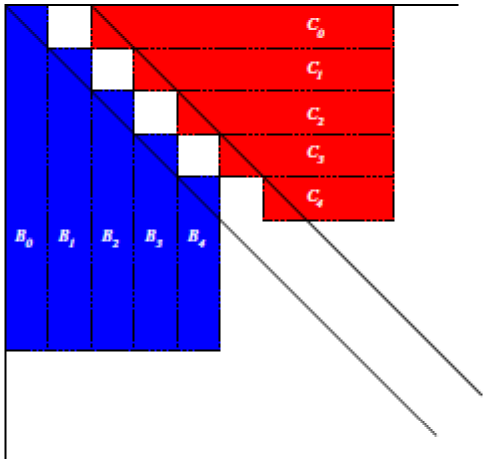


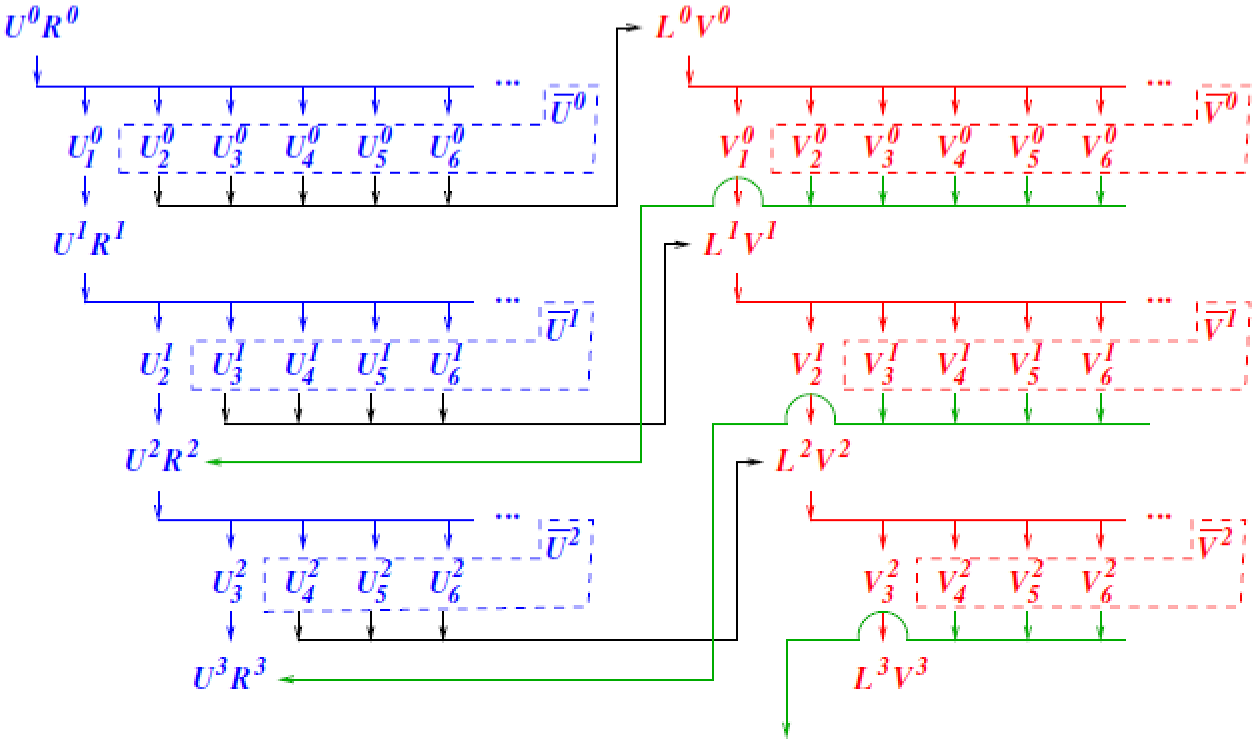


$$U^0\ R^0 \rightarrow U^0 \rightarrow L^0\ V^0 \rightarrow V^0 \rightarrow U^1R^1 \rightarrow U^1 \rightarrow L^1\ V^1 \rightarrow V^1 \ldots$$

Reduction to triangular-band for SVD

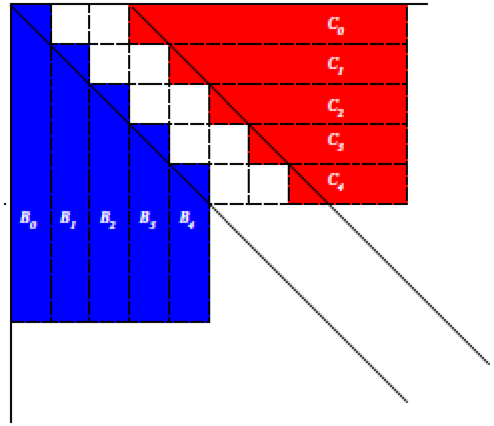
- *w=2b*

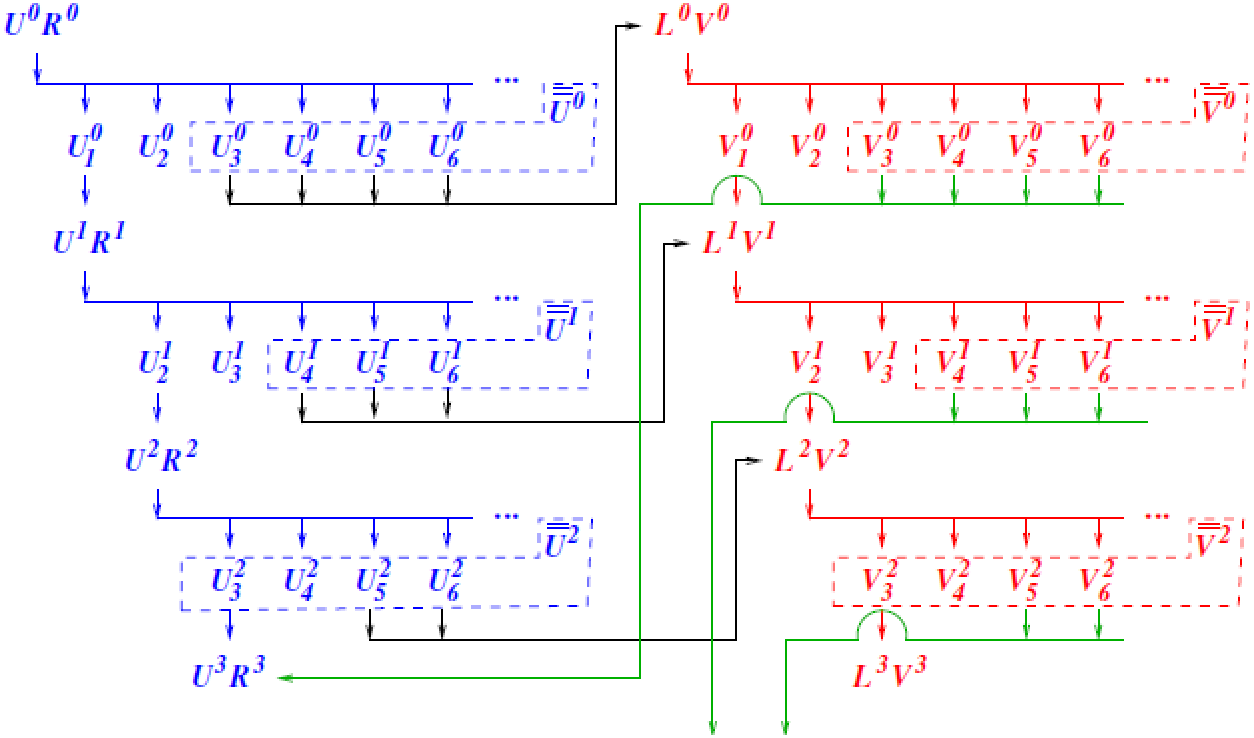




Reduction to triangular-band for SVD

- *w=3b*





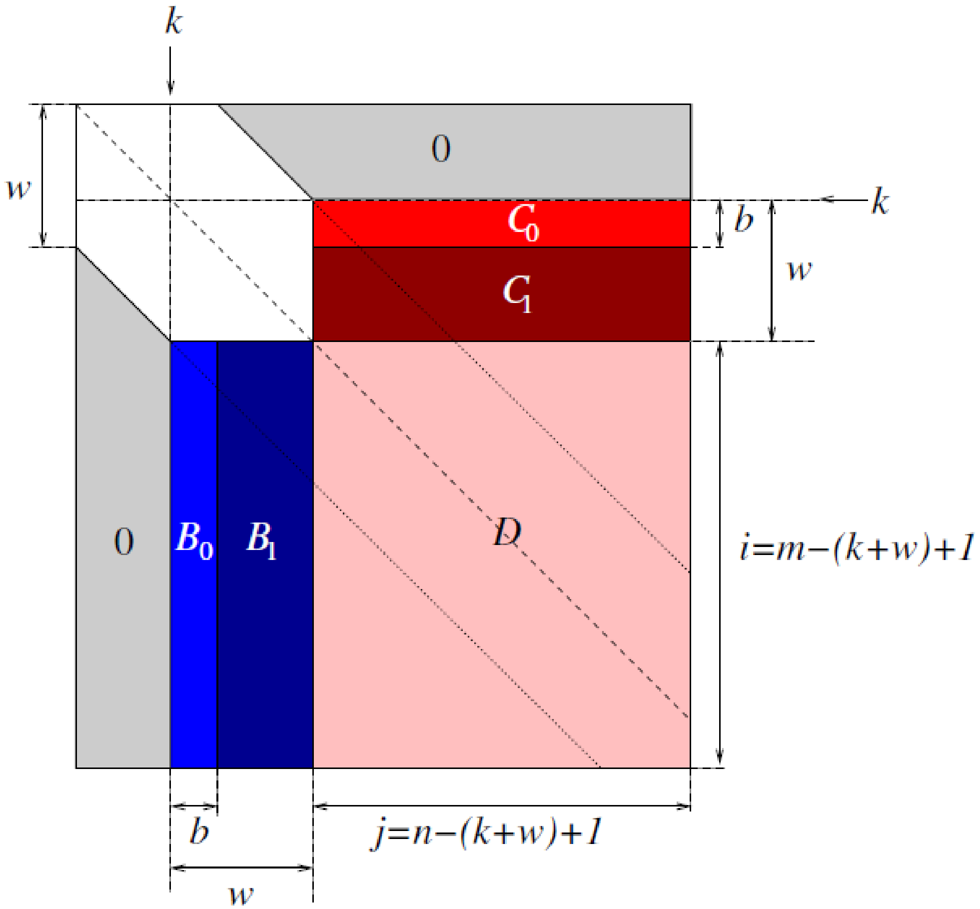
Reduction to triangular-band for SVD

- Choosing a large bandwidth *w* shifts the cost to the second stage: reduction from triangular-band to tridiagonal
- Cost of second stage is very high even for moderate *w*: bulge chasing
- A small block size *b* reduces the performance of the udpates

The restriction *3b<=w* may not be such a good idea

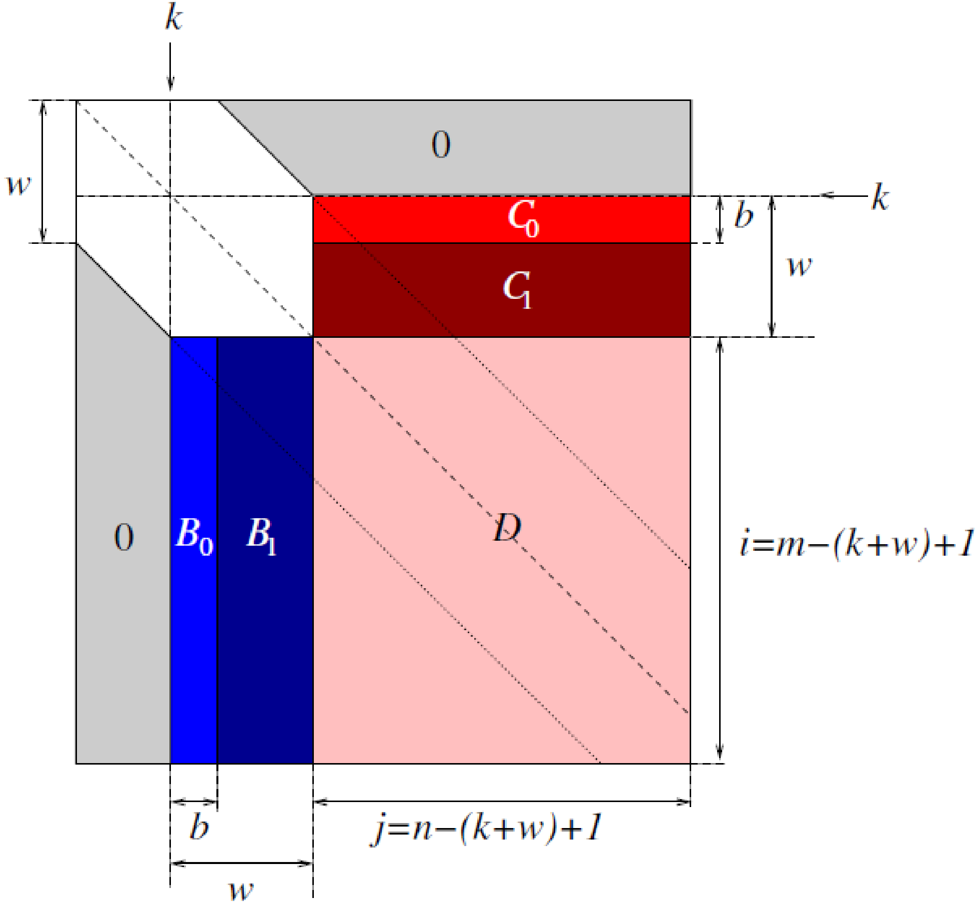
Reduction to band for SVD

- Problem arises because of overlap between *B* and *C*
  - Solution: reduce to band form



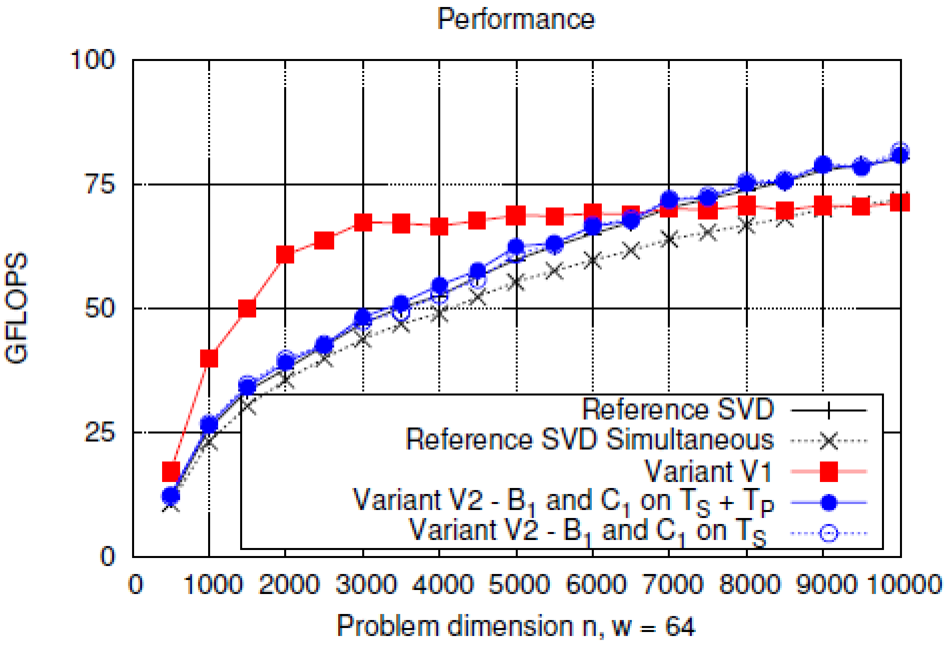
Reduction to band for SVD

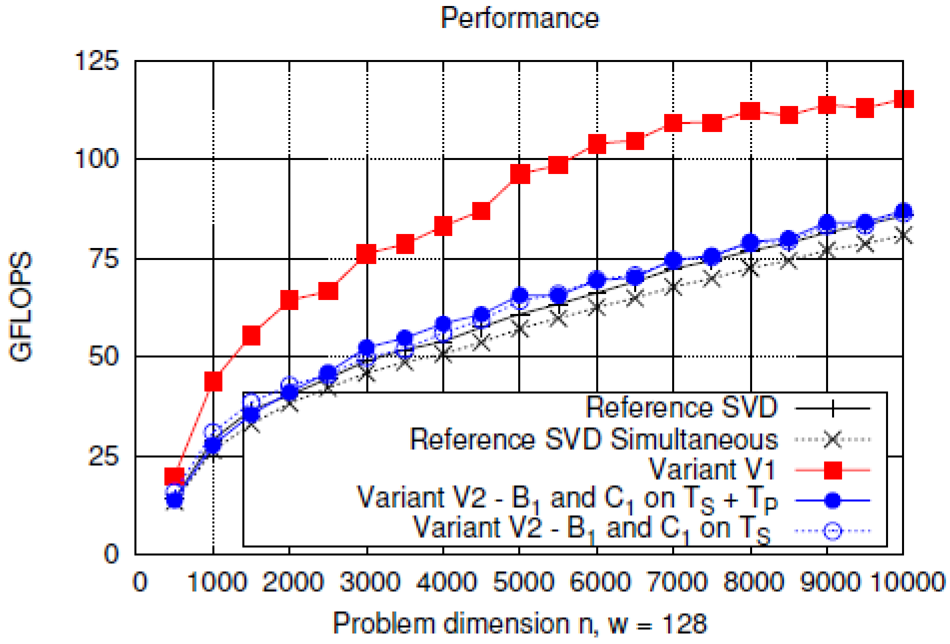
- If $2b \leq w$, next panels fall within B_1 and C_1
- No overlap. The update of these panels can be overlapped with that of D from left and right, resp.



Reduction to band for SVD

W = 64, 128





Reduction to band for SVD

- Some performance improvements:
  - In WY transform, building *W* is a Level-2 BLAS operation in the critical path:

    Employ compact WY transform instead of WY representation:

    $$Q = I - WY^T = I - YSY^T$$

  - For CPU-GPU systems, building *S* on the CPU can still be expensive and doing this operation on the GPU is not appropriate because of the fine-granularity

    Employ UT transform instead of compact WY representation:

    $$Q = I - WY^T = I - YSY^T, \text{ with } S = T^{-1}$$

    It can be built as $S = triu(YY^T)$ plus a scaling of the diagonal

Reduction to band form SEVP and SVD

- Look-ahead is possible
- With thread-level malleability, we can expect it is competitive with runtime-based approach
    - More cache-friendly than algorithms-by-blocks+runtime
- Same overhead, kernels and efficiency as standard right-looking algorithm: GPU!
- For SVP, exploit inter-iteration parallelism!

Look-ahead in Dense Matrix Factorizations

- Thanks for the attention!
- More details:

A Case for Malleable Thread-Level Linear Algebra Libraries: The LU Factorization with Partial Pivoting. S. Catalán, J. R. Herrero, R. Rodríguez-Sánchez, R. van de Geijn. https://arxiv.org/abs/1611.06365. In review in Applied Mathematics and Computation. Nov. 2016

Two-sided reduction to compact band forms with look-ahead. S. Catalán, J. R. Herrero, E. S. Quintana-Ortí, R. Rodríguez-Sánchez. A. E. Tomás. https://arxiv.org/abs/1079.00302. In review in Numerical Algorithms. July 2017