Parallel algorithms

Algorithms by blocks

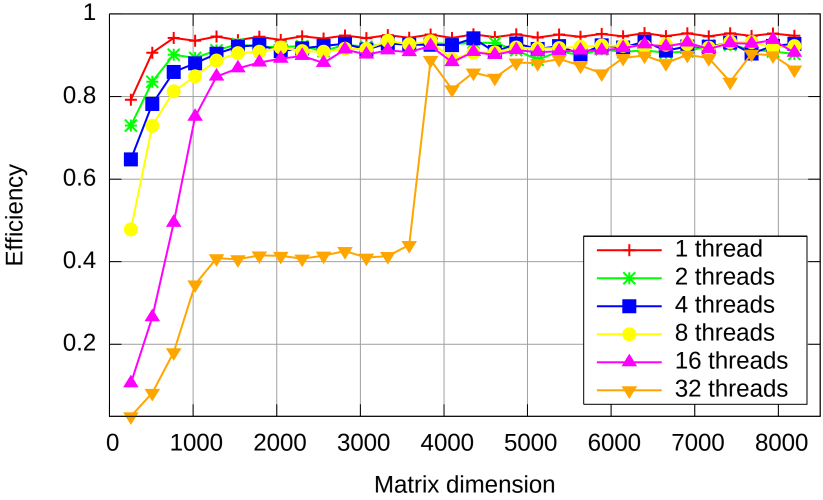
Paolo Bientinesi

Umeå Universitet
pauldj@cs.umu.se

AQTIVATE workshop
28–29 November 2023

DGEMM – “speed of light”

Efficiency: Percentage of theoretical peak performance



Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

$$\left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c|c} L_{TL}^T & L_{BL}^T \\ \hline & L_{BR}^T \end{array}\right) = \left(\begin{array}{c|c} A_{TL} & A_{BL}^T \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

$$\left(\begin{array}{c|c} L_{TL}L_{TL}^T = A_{TL} & \\ \hline L_{BL}L_{TL}^T = A_{BL} & L_{BL}L_{BL}^T + L_{BR}L_{BR}^T = A_{BR} \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad\qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Partitioned Matrix Expression (PME):

$$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & \\ \hline L_{BL} = A_{BL} L_{TL}^{-T} & L_{BR} = \Gamma\big(A_{BR} - L_{BL} L_{BL}^T\big) \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Operations:

$$\left(\begin{array}{c|c} 1)\ L_{TL} = \text{CHOL} & \\ \hline 2)\ L_{BL} = \text{TRSM} & 3)\ L_{BR} = \text{CHOL(SYRK)} \end{array}\right)$$

Cholesky Factorization

$$LL^T = A \qquad L := \Gamma(A)$$

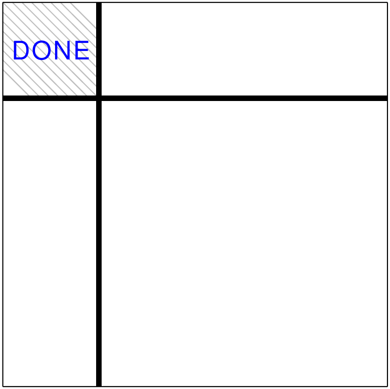
$$L = \left(\begin{array}{c|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = ?$$

Dependencies:

$$\left(\begin{array}{c|c} L_{TL} = \Gamma(A_{TL}) & \\ \hline L_{BL} = A_{BL} L_{TL}^{-T} & L_{BR} = \Gamma\big(A_{BR} - L_{BL} L_{BL}^T\big) \end{array}\right)$$

Algorithm #1

Iteration i: completed



Algorithm #1

State of the matrix:

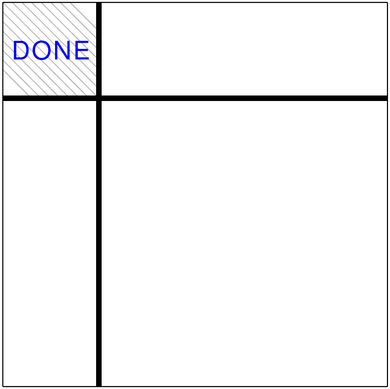
$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline & \end{array}\right)$$

Final state:

$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & L_{BR} = \mathrm{CHOL(SYRK)} \end{array}\right)$$

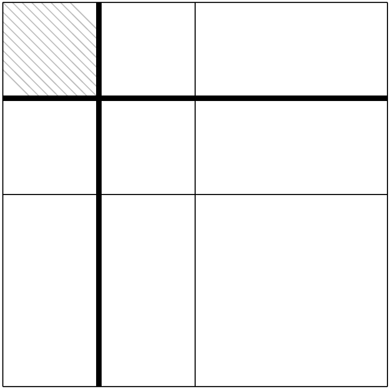
Algorithm #1

Iteration i: completed



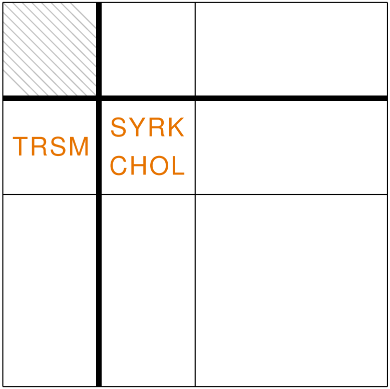
Algorithm #1

Iteration i+1: repartitioning.   Blocked vs. unblocked!



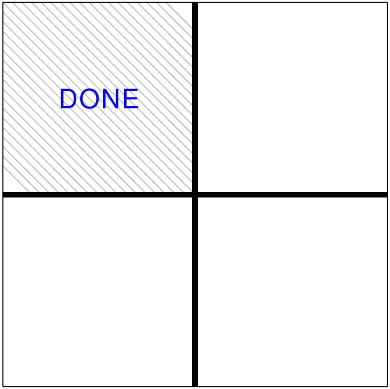
Algorithm #1

Iteration i+1: computation



Algorithm #1

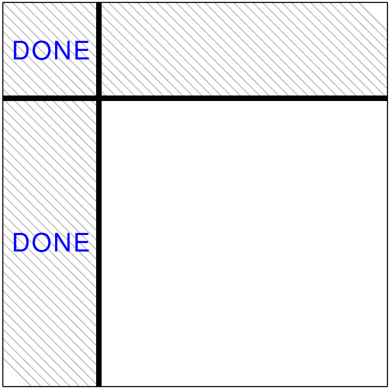
Iteration i+1: completed (boundary shift)



A Different Algorithm?

Algorithm #2

Iteration i: completed



Algorithm #2

State of the matrix:

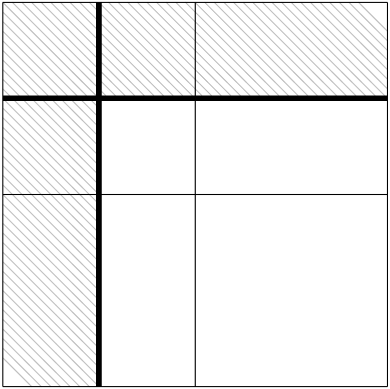
$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & \end{array}\right)$$

Final State:

$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & L_{BR} = \mathrm{CHOL(SYRK)} \end{array}\right)$$

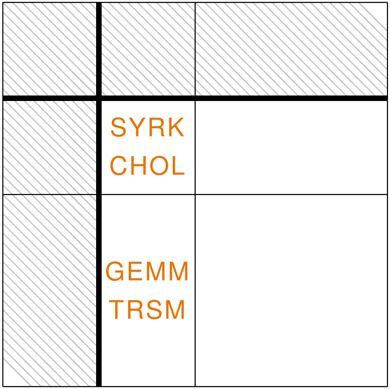
Algorithm #2

Iteration i+1: repartitioning



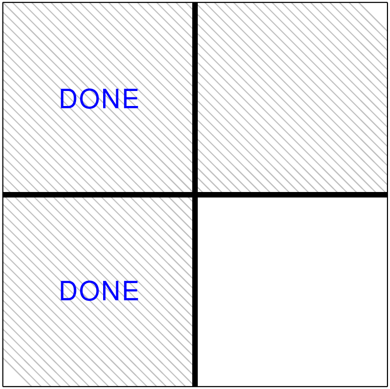
Algorithm #2

Iteration i+1: computation



Algorithm #2

Iteration i+1: completed (boundary shift)



Yet Another Algorithm!

Algorithm #3

State of the matrix:

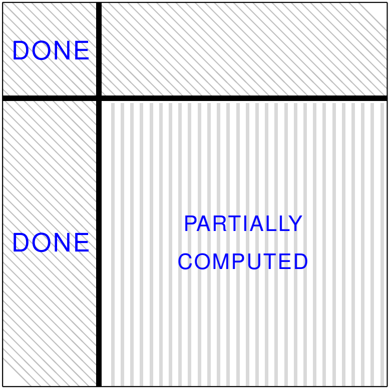
$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & L_{BR} = \mathrm{SYRK} \end{array}\right)$$

Final state:

$$\left(\begin{array}{c|c} L_{TL} = \mathrm{CHOL} & \\ \hline L_{BL} = \mathrm{TRSM} & L_{BR} = \mathrm{CHOL(SYRK)} \end{array}\right)$$

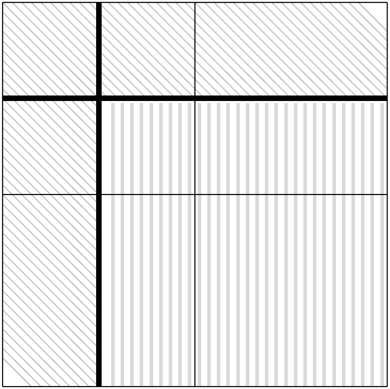
Algorithm #3

Iteration i: completed



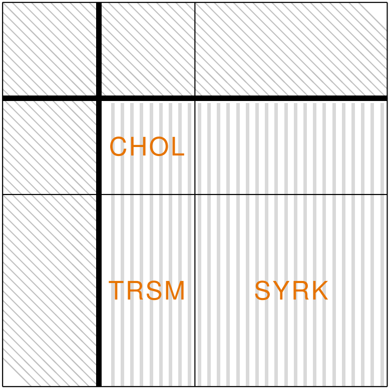
Algorithm #3

Iteration i+1: repartitioning



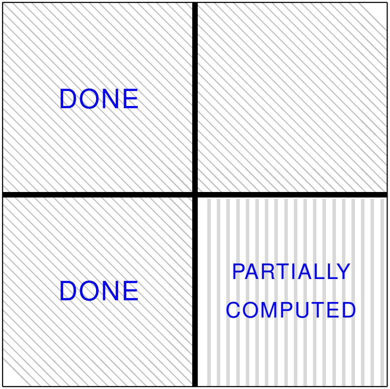
Algorithm #3

Iteration i+1: computation



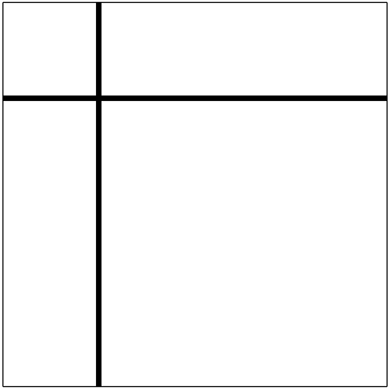
Algorithm #3

Iteration i+1: completed (boundary shift)



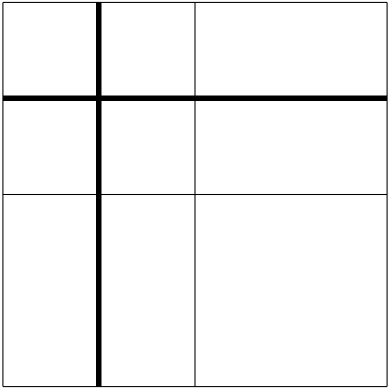
Algorithm Progression

Iteration i: completed



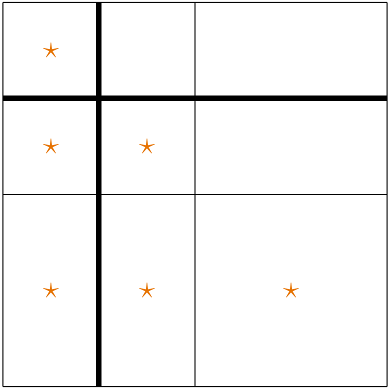
Algorithm Progression

Iteration i+1: repartitioning



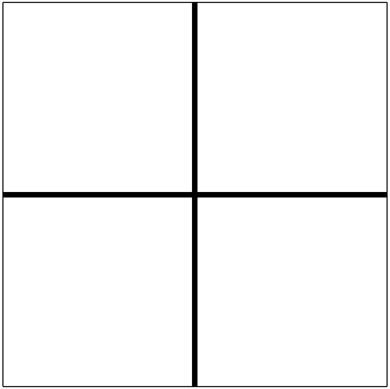
Algorithm Progression

Iteration i+1: computation



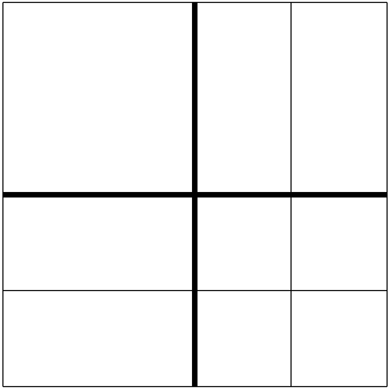
Algorithm Progression

Iteration i+1: completed (boundary shift)



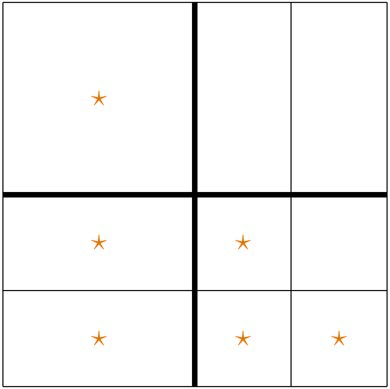
Algorithm Progression

Iteration i+2: repartitioning



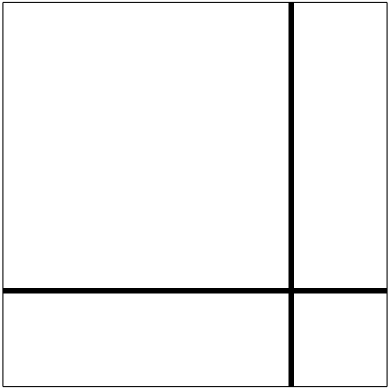
Algorithm Progression

Iteration i+2: computation



Algorithm Progression

Iteration i+2: complete (boundary shift)



Traditional code

- C, triple loop, unblocked.

```
for ( j = 0; j < n; j++ )
{
  A[j,j] = sqrt( A[j,j] );

  for ( i = j+1; i < n; i++ )
    A[i,j] = A[i,j] / A[j,j];

  for ( k = j+1; k < n; k++ )
    for ( i = k; i < n; i++ )
      A[i,k] = A[i,k] - A[i,j] * A[k,j];
}
```

Traditional code

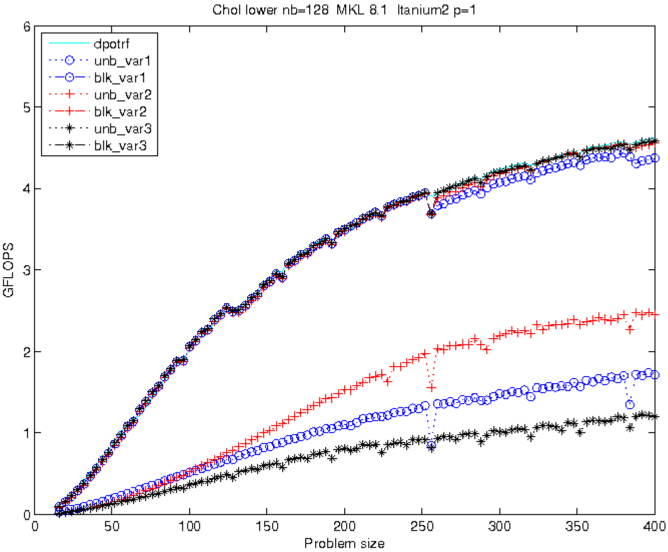
- Matlab, blocked.

```
for j = 1:nb:n,
  b = min( n-j+1, nb );

  A(j:j+b-1, j:j+b-1) = Chol( A(j:j+b-1, j:j+b-1) );

  A(j+b:n,   j:j+b-1) = A(j+b:n, j:j+b-1)/A(j:j+b-1, j:j+b-1)';

  A(j+b:n,   j+b:n  ) = A(j+b:n, j+b:n) -
                         tril(A(j+b:n, j:j+b-1)) A(j+b:n, j:j+b-1)';
end
```

Traditional code: LAPACK, blocked

```
      SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
      [..]
            DO 20 J = 1, N, NB
      *
               JB = MIN( NB, N-J+1 )
               CALL DSYRK( 'Lower', 'No transpose', JB, J-1, -ONE,
         $                 A( J, 1 ), LDA, ONE, A( J, J ), LDA )
               CALL DPOTF2( 'Lower', JB, A( J, J ), LDA, INFO )
               IF( INFO.NE.0 )
         $        GO TO 30
               IF( J+JB.LE.N-1 ) THEN
      *
                  CALL DGEMM( 'No transpose', 'Transpose', N-J-JB+1, JB,
         $                    J-1, -ONE, A( J+JB, 1 ), LDA, A( J, 1 ),
         $                    LDA, ONE, A( J+JB, J ), LDA )
                  CALL DTRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
         $                    N-J-JB+1, JB, ONE, A( J, J ), LDA,
         $                    A( J+JB, J ), LDA )
               END IF
       20   CONTINUE
```

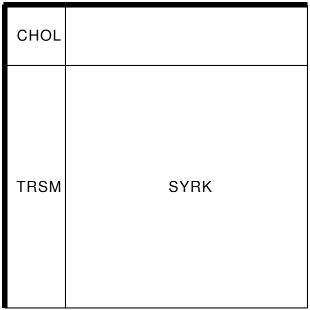
Unblocked vs. Blocked Algorithms



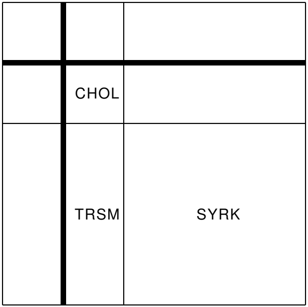
Cholesky: algorithms by blocks

Shared-memory parallelization: Can we do better?

Fork-join $\Rightarrow$ unnecessary synchronizations



Iteration 1



Iteration 2

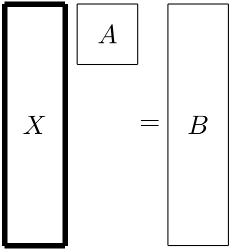
Synchronization at each iteration; in fact, at each kernel!

Shared-Memory Parallelization

- Traditional (and pipelined) parallelizations are limited by the dependencies dictated by the code.
- Parallelism should be limited only by the data dependencies.
- Idea: imitate a superscalar processor; dynamic detection of data dependencies + out of order execution.

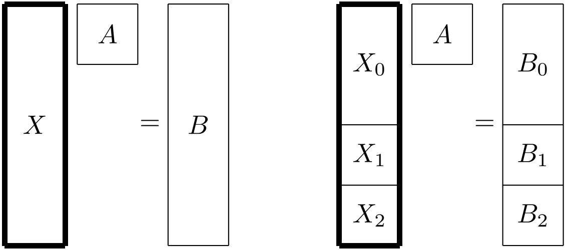
Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks



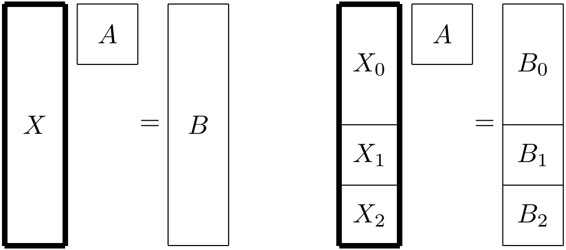
Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks



Back to Cholesky: How to create parallelism?

- Idea: decompose the tasks



$$X_0 A = B_0$$

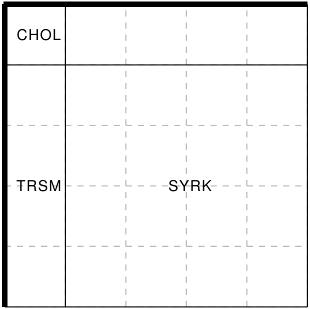
$$X_1 A = B_1$$

$$X_2 A = B_2$$

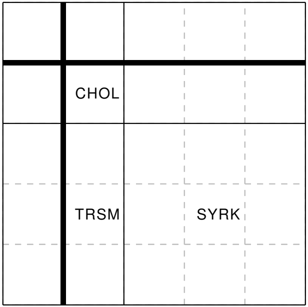
Algorithms by blocks

Also, “Tiled algorithms”. Not “blocked”!

Goal: Create small tasks, feed all processors as early as possible



Iteration i



Iteration i+1

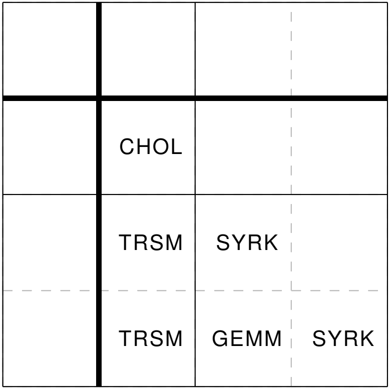
Breaking down the computation

Decomposition in tiles (iteration 1)

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

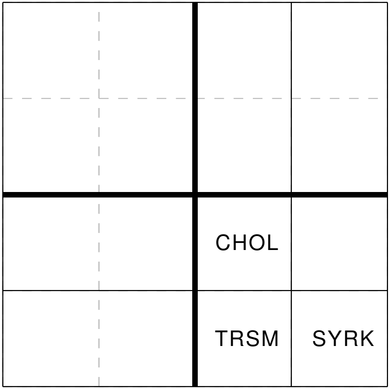
Breaking down the computation

Decomposition in tiles (iteration 2)



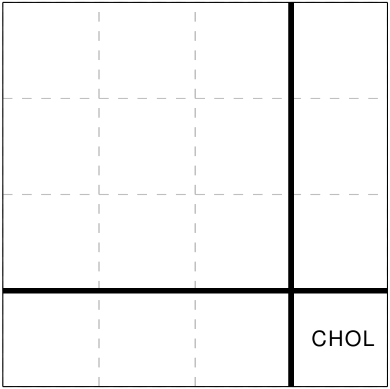
Breaking down the computation

Decomposition in tiles (iteration 3)



Breaking down the computation

Decomposition in tiles (iteration 4)



Dependencies

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

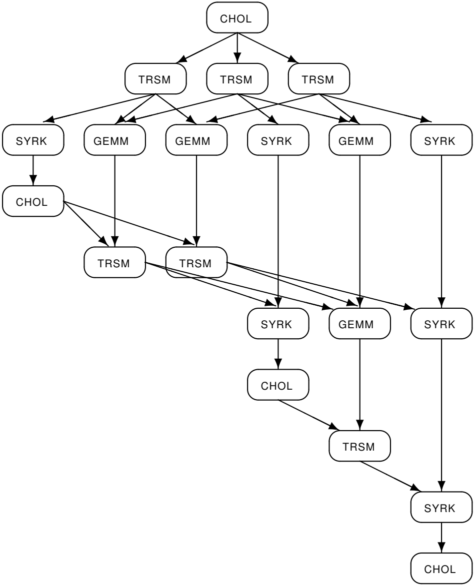
CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

Dependencies

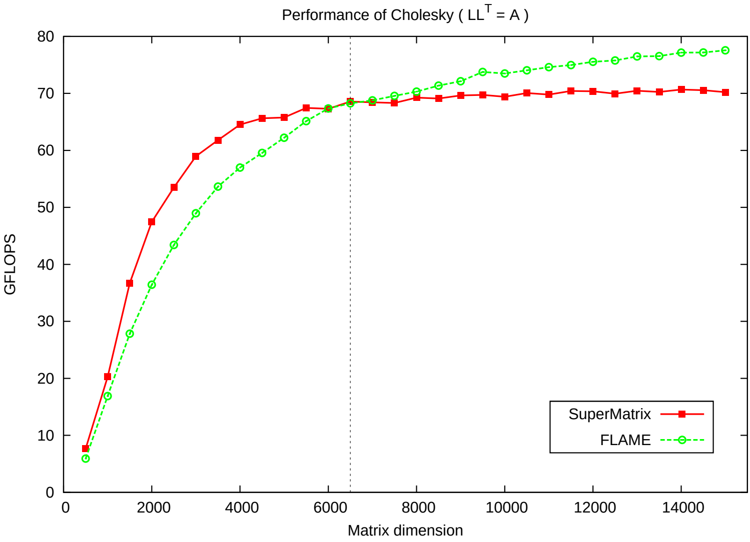
CHOL			
TRSM	SYRK		
TRSM	GEMM	SYRK	
TRSM	GEMM	GEMM	SYRK

DAG - Dependencies

4×4-tile matrix



Crossover, 16 cores



Runtime systems, e.g., SuperMatrix, StarPU, Quark, . . .

Taskqueues

- The runtime system “pre-executes” the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, . . .

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system "pre-executes" the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, . . .

Taskqueues

- The runtime system “pre-executes” the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.
- Upon termination of a task, the thread notifies dependent tasks and updates the queue.

Runtime systems, e.g., SuperMatrix, StarPU, Quark, ...

Taskqueues

- The runtime system “pre-executes” the code.
  Whenever a kernel is encountered, one or more tasks are created and inserted in a global task queue.
- Dependencies between tasks are calculated, forming a DAG.
- Tasks with all input operands available are executable.
  The other tasks must wait in the queue.
- Threads asynchronously dequeue tasks from the queue.
- Upon termination of a task, the thread notifies dependent tasks and updates the queue.
- Loop until all tasks complete execution.

Task Execution

5×5-tile matrix

Stage	Scheduled Tasks			
1	CHOL			
2	TRSM	TRSM	TRSM	TRSM
3	SYRK	GEMM	SYRK	GEMM
4	GEMM	SYRK	GEMM	GEMM
5	GEMM	SYRK	CHOL	
6	TRSM	TRSM	TRSM	
7	SYRK	GEMM	SYRK	GEMM
8	GEMM	SYRK	CHOL	
9	TRSM	TRSM		
10	SYRK	GEMM	SYRK	
11	CHOL			
12	TRSM			
13	SYRK			
14	CHOL			

SPD Inv: 1) Chol 2) Inv 3) Mat Mat Mult.

5×5-tile matrix

$$A := A^{-1}$$

$$A := \left(LL^T\right)^{-1}$$

$$A := L^{-T} L^{-1}$$

SPD Inv: 1) Chol 2) Inv 3) Mat Mat Mult.

5×5-tile matrix

Stage	Scheduled Tasks			
1	CHOL			
2	TRSM	TRSM	TRSM	TRSM
3	SYRK	GEMM	SYRK	GEMM
4	GEMM	SYRK	GEMM	GEMM
5	GEMM	SYRK	CHOL	TRSM
6	TRSM	TRSM	TRSM	TRSM
7	TRSM	TRSM	TRINV	SYRK
8	GEMM	SYRK	GEMM	GEMM
9	SYRK	TTMM	CHOL	TRSM
10	TRSM	TRSM	TRSM	TRSM
11	GEMM	GEMM	GEMM	SYRK
12	GEMM	SYRK	TRSM	CHOL
13	TRSM	TRSM	TRINV	SYRK
14	TRSM	GEMM	GEMM	GEMM
15	GEMM	TRMM	SYRK	TRSM
16	TRSM	TTMM	CHOL	TRSM
17	SYRK	TRINV	GEMM	SYRK
18	GEMM	GEMM	GEMM	TRMM
19	TRMM	TRSM	TRSM	TRSM
20	TRSM	TRSM	TRSM	TRSM
21	TTMM	SYRK	GEMM	SYRK
22	TRINV	GEMM	GEMM	TRINV
23	SYRK	SYRK	GEMM	SYRK
24	TRMM	GEMM	TRMM	GEMM
25	TRMM	SYRK	GEMM	GEMM
26	TTMM	GEMM	TRMM	TRMM
27	SYRK	TRMM		
28	TRMM			
29	TTMM			

Not quite so simple

Not quite so simple

- Storage by blocks

Not quite so simple

- Storage by blocks
- Critical path

Not quite so simple

- Storage by blocks
- Critical path
- Cache “simulator”

Not quite so simple

- Storage by blocks
- Critical path
- Cache “simulator”
- Tension between size of blocks and number of blocks

Multithreaded BLAS vs. Algorithms-by-blocks

No absolute winner: crossover!

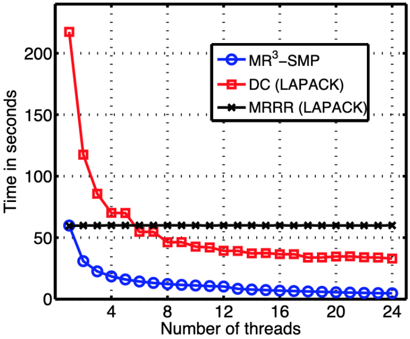
- ✓ Ease of use
- ✖ Synchronization

- ✓ Out of order execution
- ✓ Parallelism dictated by data dependencies
- ✖ Plateaux

How (NOT) to present parallel results

How (NOT) to present parallel results

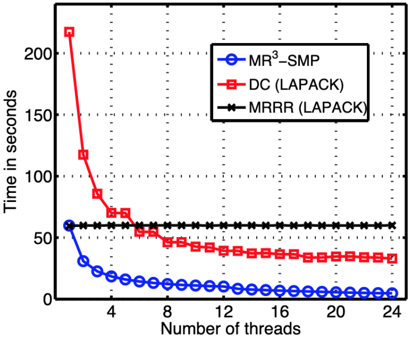
#procs vs. time

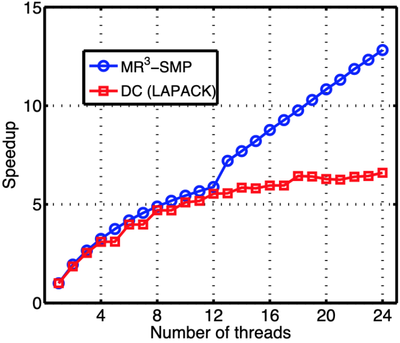


Does blue scale better or worse than red?
How much better or worse? Can you tell?

How (NOT) to present parallel results

#procs vs. time

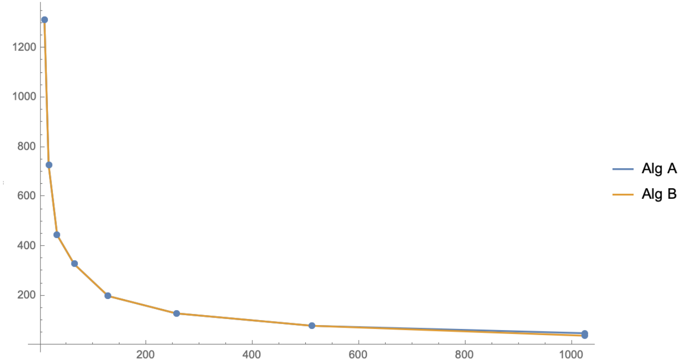




Does blue scale better or worse than red?

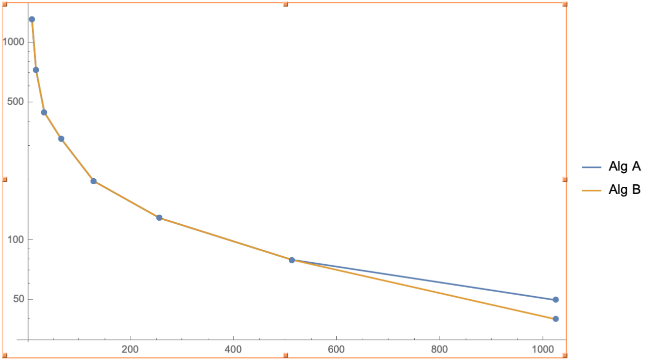
How much better or worse? Can you tell?

#procs vs. time



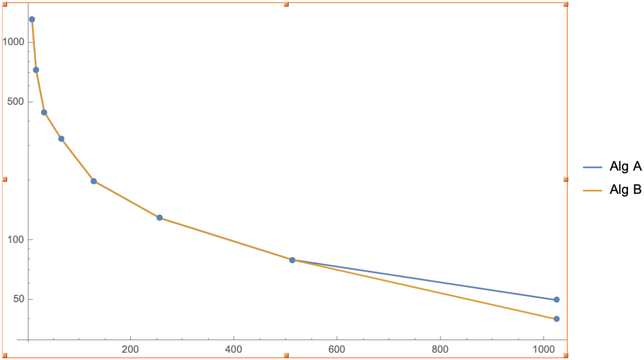
At 1024 procs, Alg B is 20% faster than Alg A.
Do you see it?

#procs vs. time, log scale



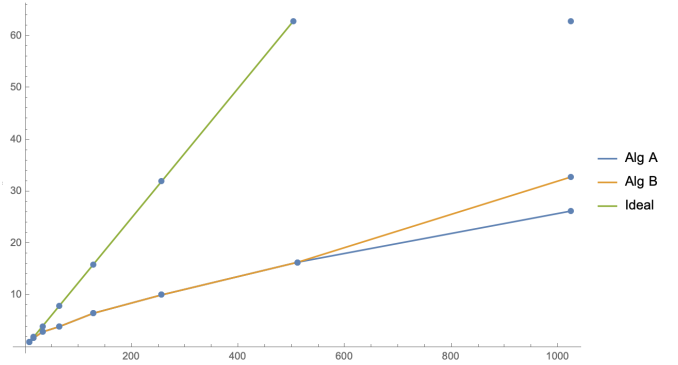
At 1024 procs, Alg B is 20% faster than Alg A.
Do you see it?

#procs vs. time, log scale



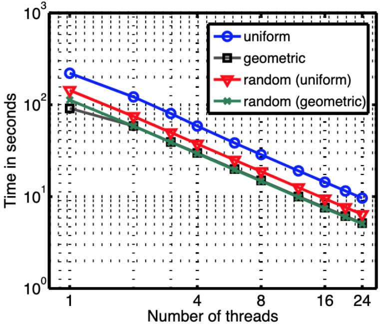
How well do Alg A and Alg B scale?
Can you tell?

speedup



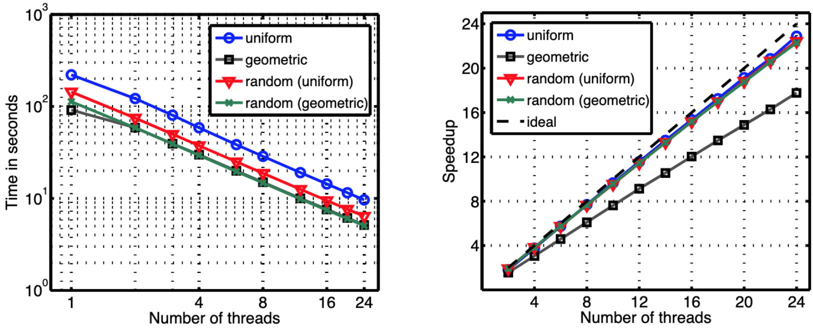
How well do Alg A and Alg B scale?
Can you tell?

#procs vs. time, log scale



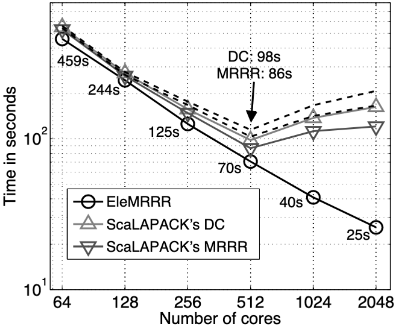
Does “geometric” scale bettwer/worse than the others?

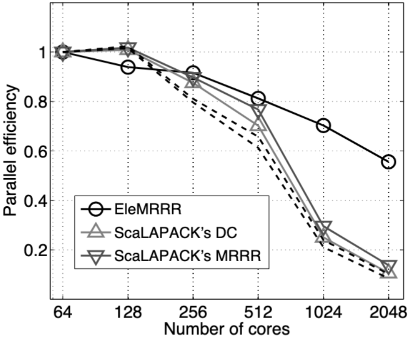
#procs vs. time, log scale



Does “geometric” scale bettwer/worse than the others?

strong scalability

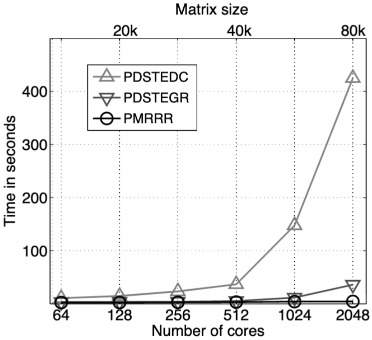




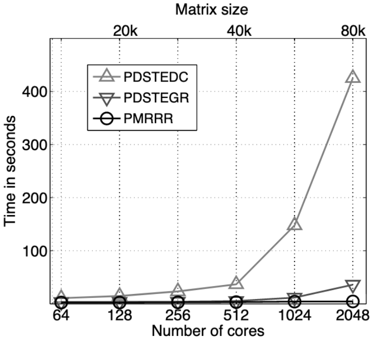
Overhead blow up

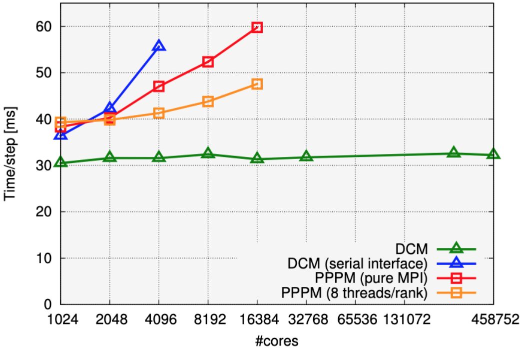
Efficiency greater than 1

weak scalability

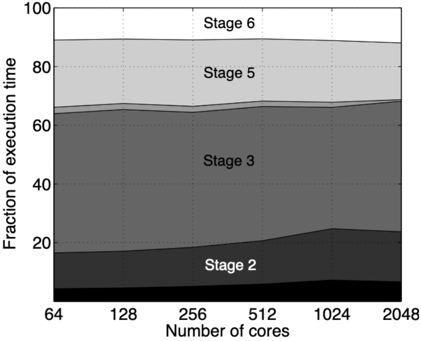


weak scalability

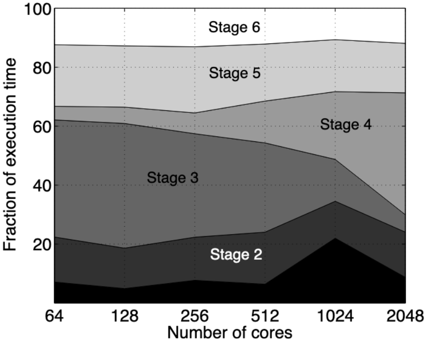




weak scalability



all stages scale $\Rightarrow$ the alg. scales



the alg. does not scale