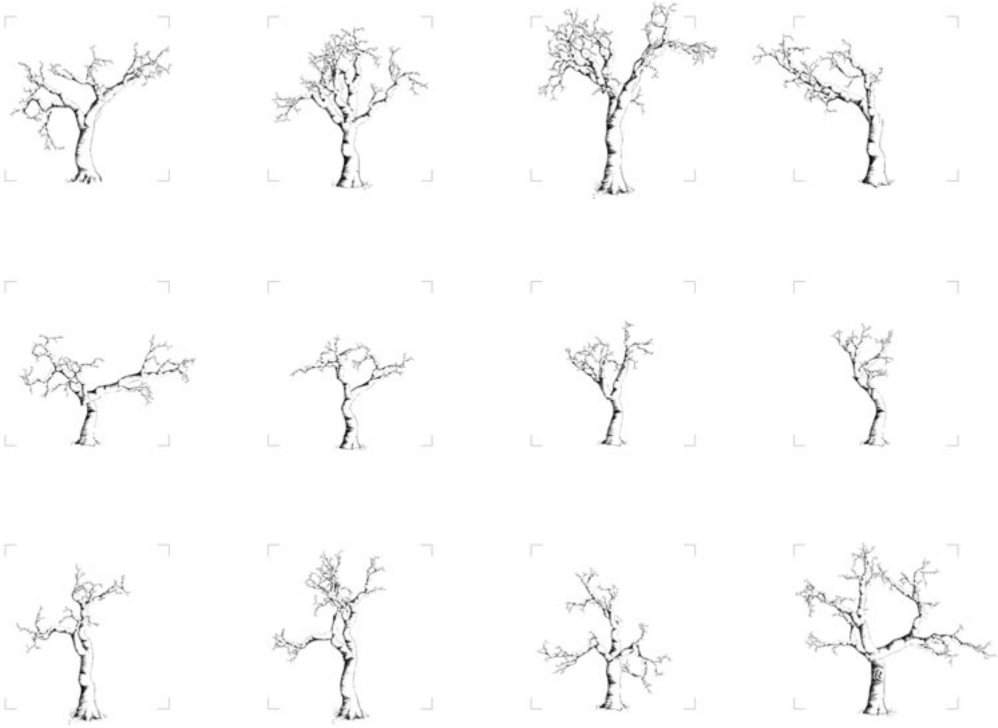
Cholesky inversion, DAGs, critical path, with some scheduling

Julien Langou, University of Colorado Denver.

Thursday June 3rd, 2010.



Many thanks to the students: Henricus Bouwmeester and Cédric Bourrasset.

Cholesky inversion, DAGs, critical path, with some scheduling

Our new mission, today:

starting from A, an n–by–n symmetric positive definite matrix, compute A^{-1}

(A^{-1} is the inverse of A).

Mission cancelled?

> "*In the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute* A^{-1}." — Forsythe, Malcolm and Moler

Mission reengaged ...
The computation of the variance-covariance matrix in statistics requires that the inverse is calculated explicitly.

Methodology: Algorithm by tiles. (Of course!)

Criteria for selection: We keep the granularity of our algorithms constant (the block size), then an algorithm is said to be better than another if it has a shorter critical path. We count the length in term of tasks (as opposed to flops).

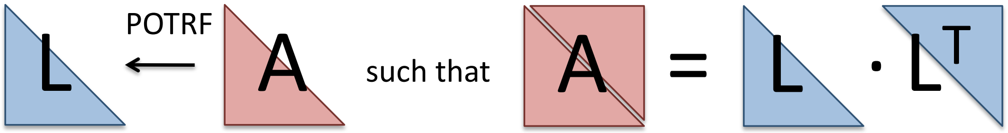
Related Work: The final algorithm we recommend has already been recommended by Chan and al., SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks, *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008. See the libflame sofware as well.

Cholesky inversion, DAGs, critical path, with some scheduling

To find the inverse of a symmetric positive definite matrix, A, the standard method is to

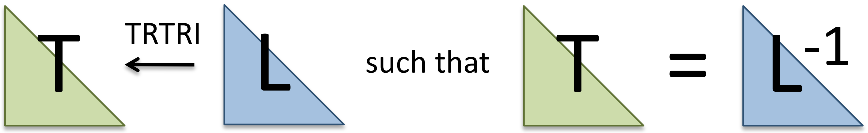
1) **POTRF** compute the Cholesky factorization (lower triangular)

$$\text{compute lower triangular } C \text{ such that } A = CC^T$$



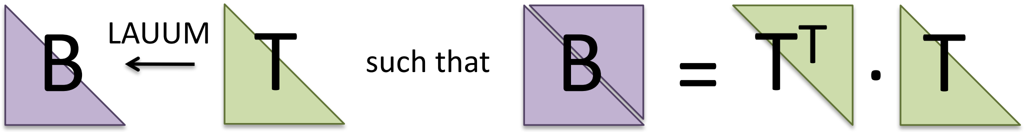
2) **TRTRI** compute the inverse of the Cholesky factor

$$\text{compute } T \text{ such that } T = C^{-1}$$

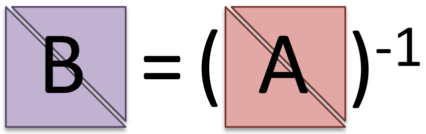


3) **LAUUM** multiply T^T and T

$$\text{compute (half of) } A^{-1} \text{ such that } T^T T = \left(LL^T\right)^{-1} = A^{-1}$$

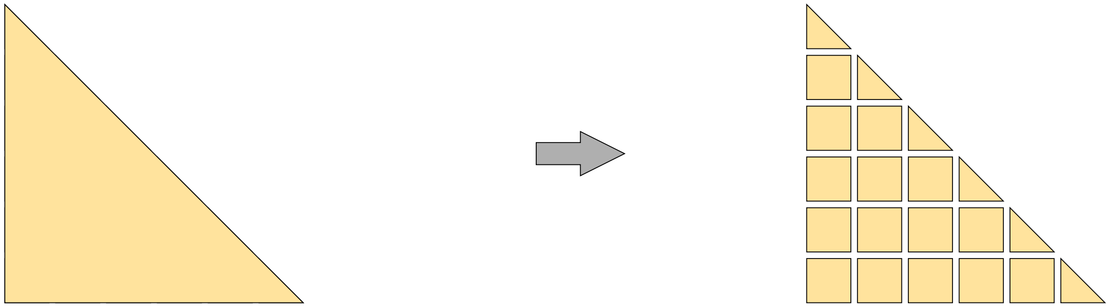


Note: all this is done **in place** in practice.
Please check that



Cholesky inversion, DAGs, critical path, with some scheduling

In a new family of algorithms, the *tile algorithms*, the approach is to "split" a n–by–n matrix into t–by–t tiles and consider each tile as a scalar. Since we are considering symmetric matrices, we will only concern ourselves with the lower portion (although the upper portion would be just as valid).

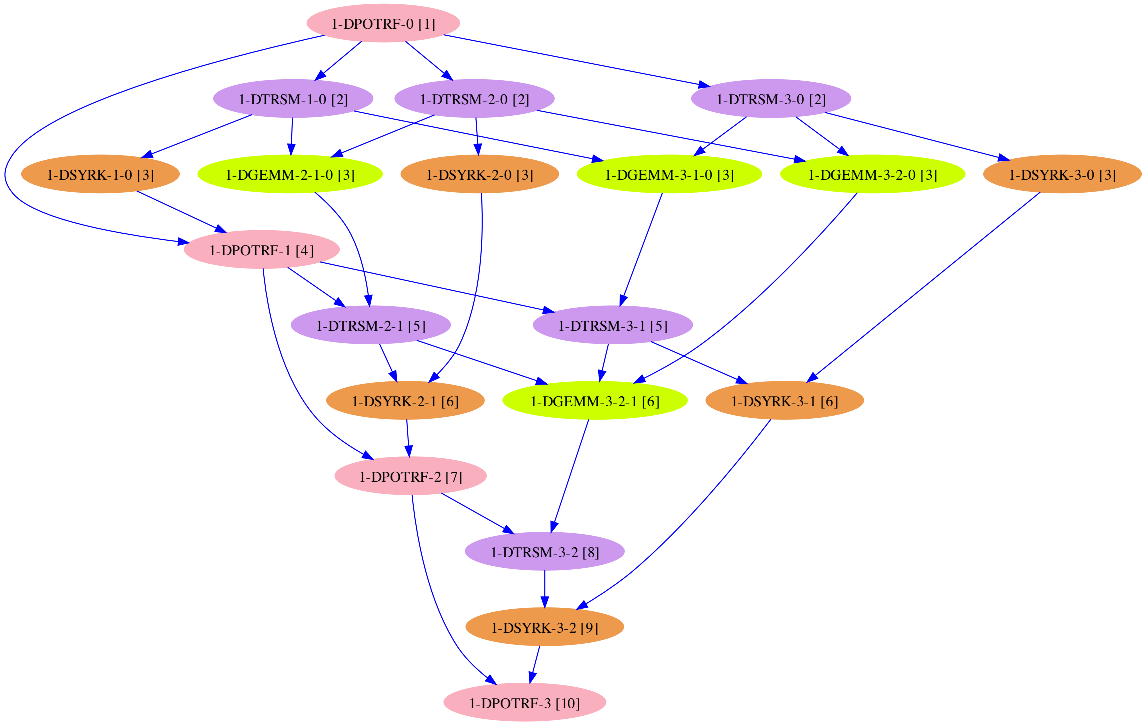


Previous research has shown that it is possible to write efficient and scalable tile algorithms for performing a Cholesky factorization, a (pseudo) LU factorization, a QR factorization, and computing the inverse of a symmetric positive definite matrix!

Cholesky inversion, DAGs, critical path, with some scheduling

For the first step in our inversion, we apply the Cholesky factorization to our tiles producing the following *directed acyclic graph* (DAG). Within the DAG, each node represents the fine granularity tasks in which the operation can be decomposed and the edges represent the dependencies among them.

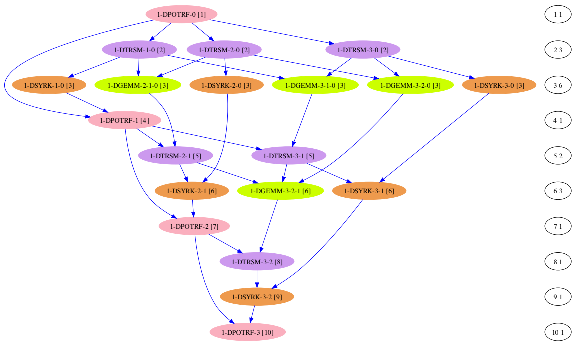
```
Step 1: Tile Cholesky Factorization (compute I
$A = LL^T$);
for $j = 1$ to $t$ do
    for $k = 1$ to $j$ do
        $A_{j,j} \leftarrow A_{j,j} - A_{j,k} * A^T_{j,k}$ (SYRK(j,k)) ;
    end
    $A_{j,j} \leftarrow CHOL(A_{j,j})$ (POTRF(j)) ;
    for $i = j+1$ to $t$ do
        for $k = 1$ to $j$ do
            $A_{i,j} \leftarrow A_{i,j} - A_{i,k} * A^T_{j,k}$ (GEMM(i,j,k))
        end
    end
    for $i = j+1$ to $t$ do
        $A_{i,j} \leftarrow A_{i,j} / A^T_{j,j}$ (TRSM(i,j)) ;
    end
end
```



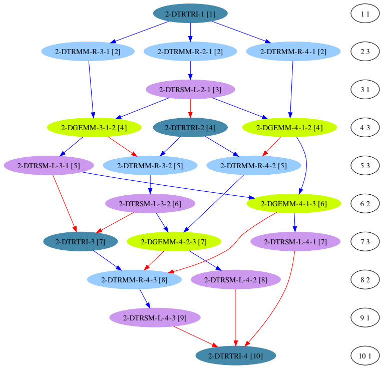
It is relatively straight forward to translate an algorithm from LAPACK to obtain a ready-to-be-executed tile algorithm. Note that for the right-looking and bordered variant of the Cholesky factorization (POTRF), we obtain the same DAG.

Cholesky inversion, DAGs, critical path, with some scheduling

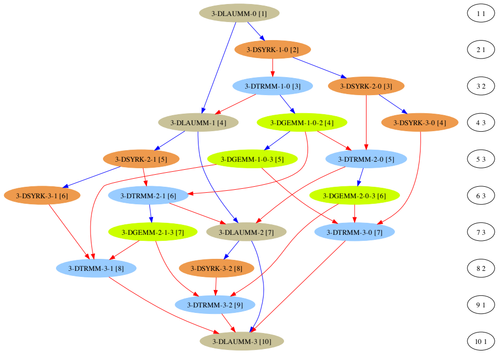
To perform the matrix inversion, we can apply this same method to each step. Thus the three DAGs representing each step in the inversion for a matrix composed of four tiles could be



POTRF : L ← chol(A)
(Step 1)



TRTRI : $\mathrm{T} \leftarrow \mathrm{L}^{-1}$
(Step 2)



LAUUM : $\mathrm{A}^{-1} \leftarrow \mathrm{T}^T\mathrm{T}$
(Step 3)

The operations are performed in-place, i.e., the data is overwritten. Note that different variants of the same operation may produce different DAGs.

Cholesky inversion, DAGs, critical path, with some scheduling

Criteria for selection: We keep the granularity of our algorithms constant (the block size), then an algorithm is said to be better than another if it has a shorter critical path. We count the length in term of tasks (as opposed to flops).

How to change the critical path?

POTRF: Almost[1] no way. Game over! ([1]: associativity and the commutativity of the GEMMs)

Upper and Lower: No diversity here as well.

1. Mathematical formulation of algorithm

- **TRTRI:** Algorithmic variants can produce different DAGs resulting in shorter critical paths and/or better pipelining. For example, variants for TRTRI are dependent upon whether we compute T (which we want to be L^{-1} such that
$$LT = I \quad \text{or} \quad TL = I.$$
(Since $LL^{-1} = L^{-1}L = I$, both derivations are valid.)
- **TRTRI and LAUUM:** The in-place formulation is like a “shell game”. There are a lot of formulation possible.
- **POTRF, TRTRI and LAUUM:** Commutativity and associativity within the GEMMs may be used to alter the dependencies in such a way to reduce the critical path.

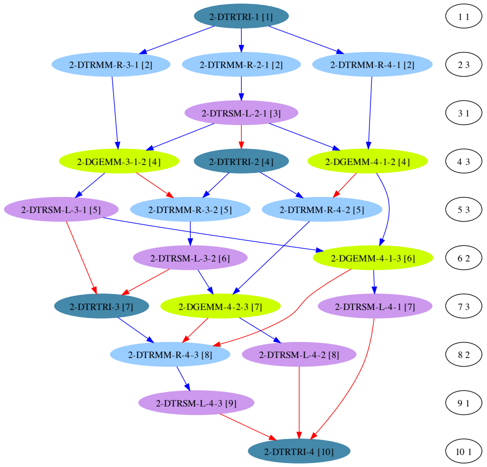
2. Compiler technology

- **POTRF+TRTRI+LAUUM:** Instead of performing the steps sequentially, we pipeline (or interleave) the entire algorithm. Note: we will see that the best variants for POTRF+TRTRI+LAUUM is not POTRF, followed by the best variant for TRTRI, followed by the best variant for LAUUM!
- **TRTRI and LAUUM:** Substituting out-of-place for in-place algorithms removes anti-dependencies (write-after-read) through the use of temporary buffers (array renaming).

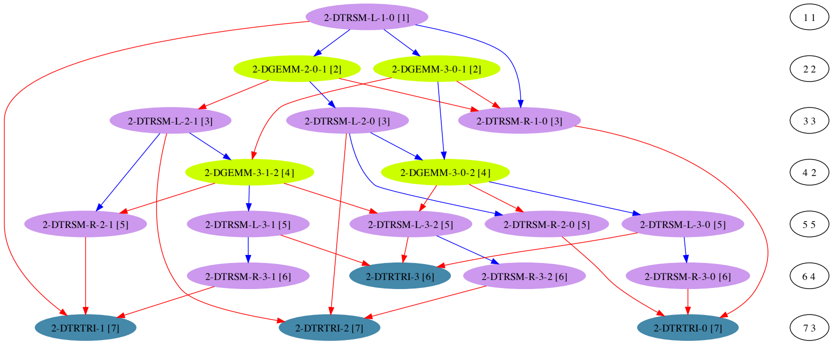
Cholesky inversion, DAGs, critical path, with some scheduling

We have analyzed one variant for POTRF, six variants for TRTRI and three variants for LAUUM. (Many more variants exists!)

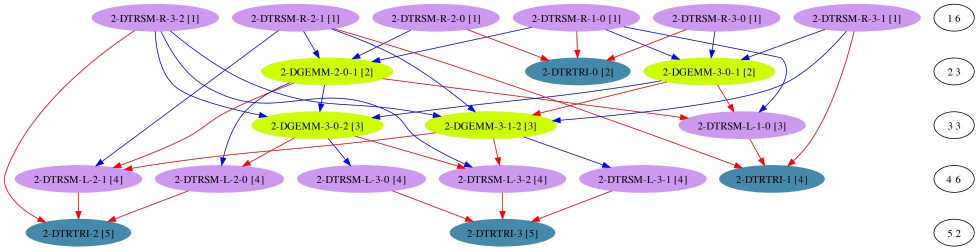
Here are nice pictures for variants of TRTRI with $t = 4$. We play different shell games with in place.



Variant 1: Critical Path = 10 (=3t-2)



Variant 2: Critical Path = 7 (=2t-1)



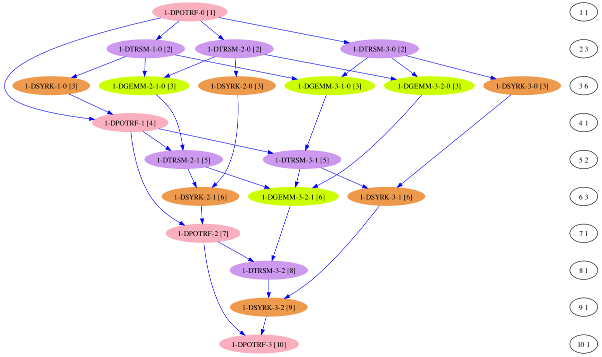
Variant 3: Critical Path = 5 (=t+1)

Variant 4 (resp. 5 and 6) has the same DAG as variant 1 (resp. 2 and 3) but instead of starting from top left corner and ending bottom right, it starts from bottom right and ends up top left. It is obtained from considering $LL^{-1} = I$ instead of $L^{-1}L = I$.

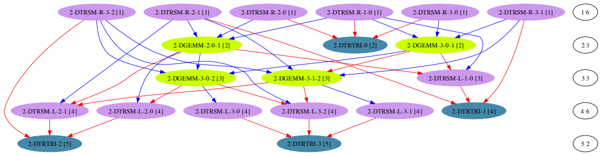
Cholesky inversion, DAGs, critical path, with some scheduling

We have analyzed one variant for POTRF, six variants for TRTRI and three variants for LAUUM.
(Many more variants exists!)
If we perform the steps for the inversion in a sequential manner, then given the following results we would choose POTRF, TRTRI_v3 (or v6) and LAUUM_v1 (or v2 or v3) and settle with $7t-3$.

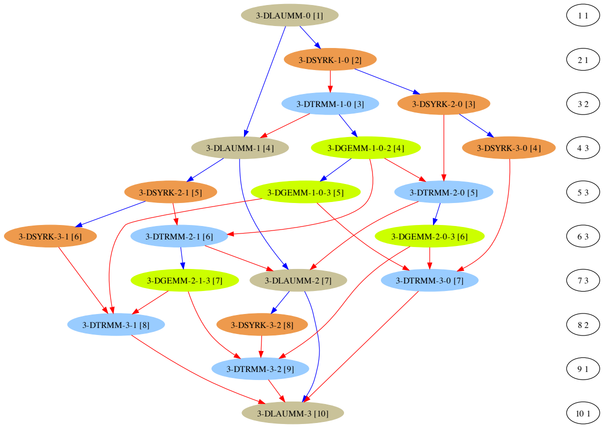
Algorithm	Variant	CP length
POTRF		3t-2
TRTRI	1	3t-2
	2	2t-1
	3	t+1
	4	3t-2
	5	2t-1
	6	t+1
LAUUM	1	3t-2
	2	3t-2
	3	3t-2



POTRF: $3t-2$



TRTRI_v3: $t+1$

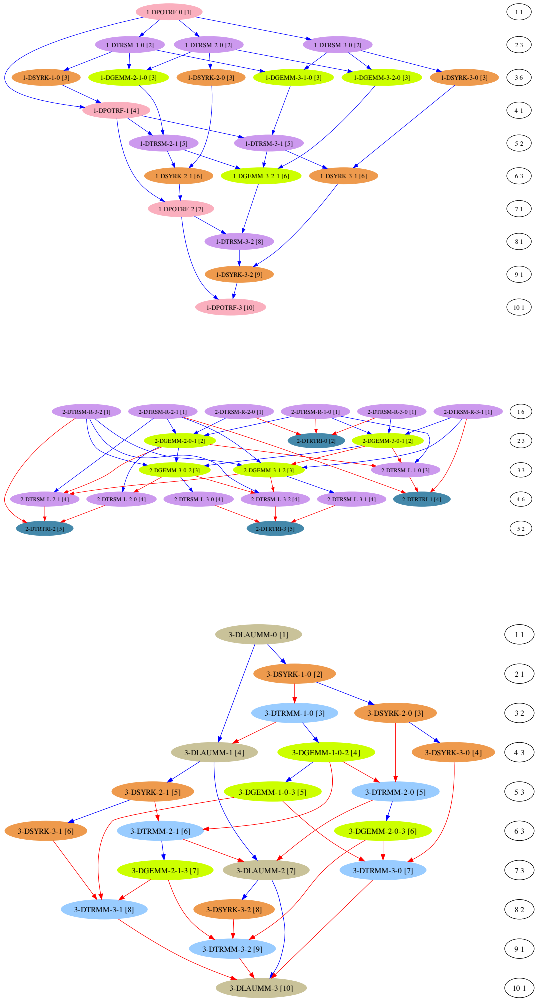


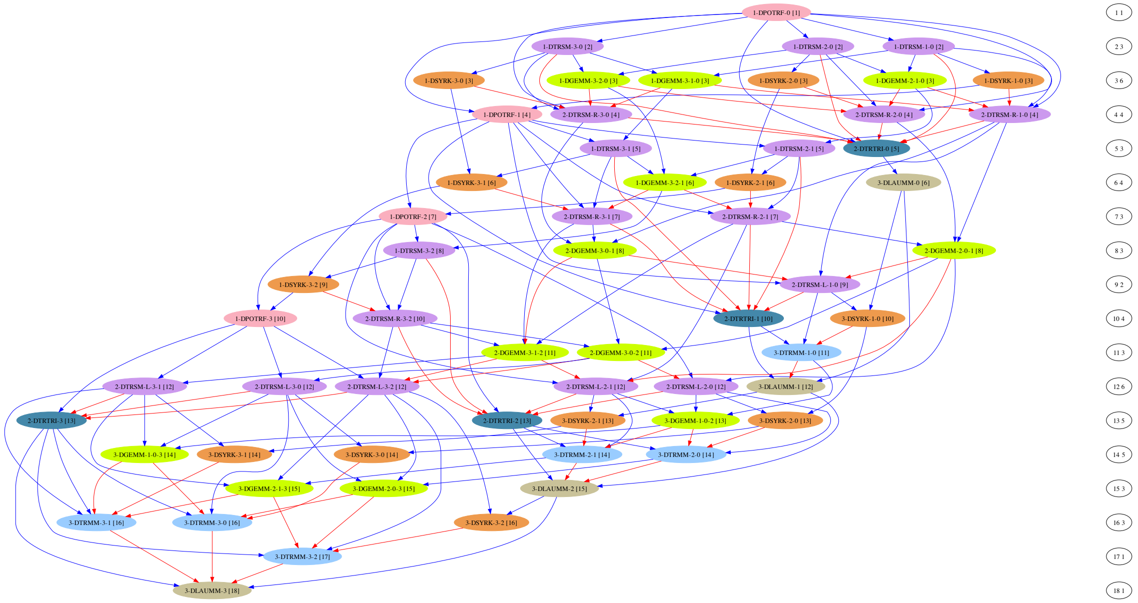
LAUUM_v1: $3t-2$

Total	$7t-3$

Cholesky inversion, DAGs, critical path, with some scheduling

By pipelining (or interleaving) the multiple steps of the inversion algorithm, we can reduce the length of the critical path even further. Taking POTRF, TRTRI_v3 and LAUUM_v1, we obtain $3t + 6$.



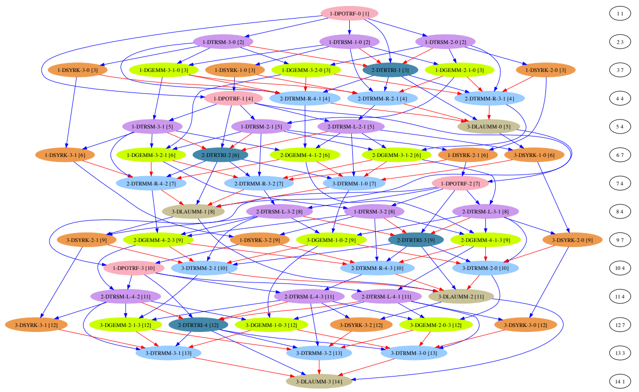


Sequential(25): $7t - 3$

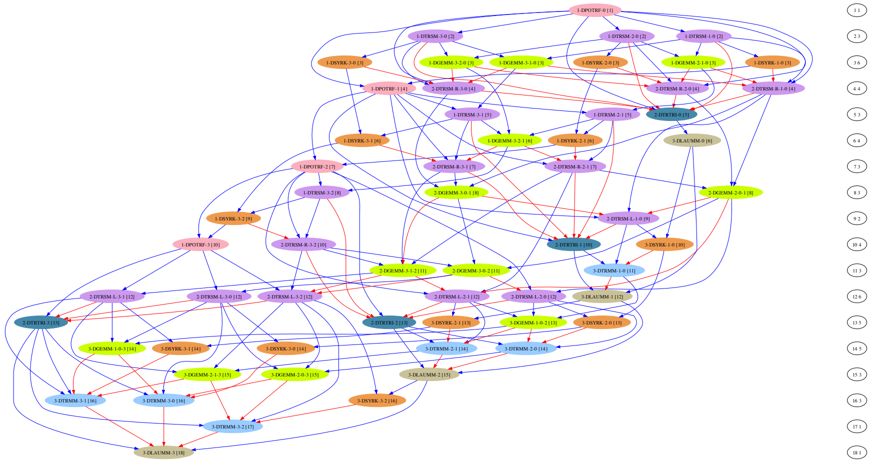
Pipelined(18): $3t + 6$

Cholesky inversion, DAGs, critical path, with some scheduling

Consider that taking POTRF, TRTRI_v1 and LAUUM_v1 sequentially provides us with a ciritcal path of length $9t-6$, but the pipelined version has a critical path of length $3t+2$ which is shorter than our previous combination.



(14): $3t+2$ (POTRF+TRTRIv1+LAUUMv1)



(18): $3t+6$ (POTRF+TRTRIv3+LAUUMv1)

Note: Cholesky has a critical path of $3t-2$ therefore the whole inversion ($3t+2$) is obtained for "four additional tasks more". This is quite a feat![2].

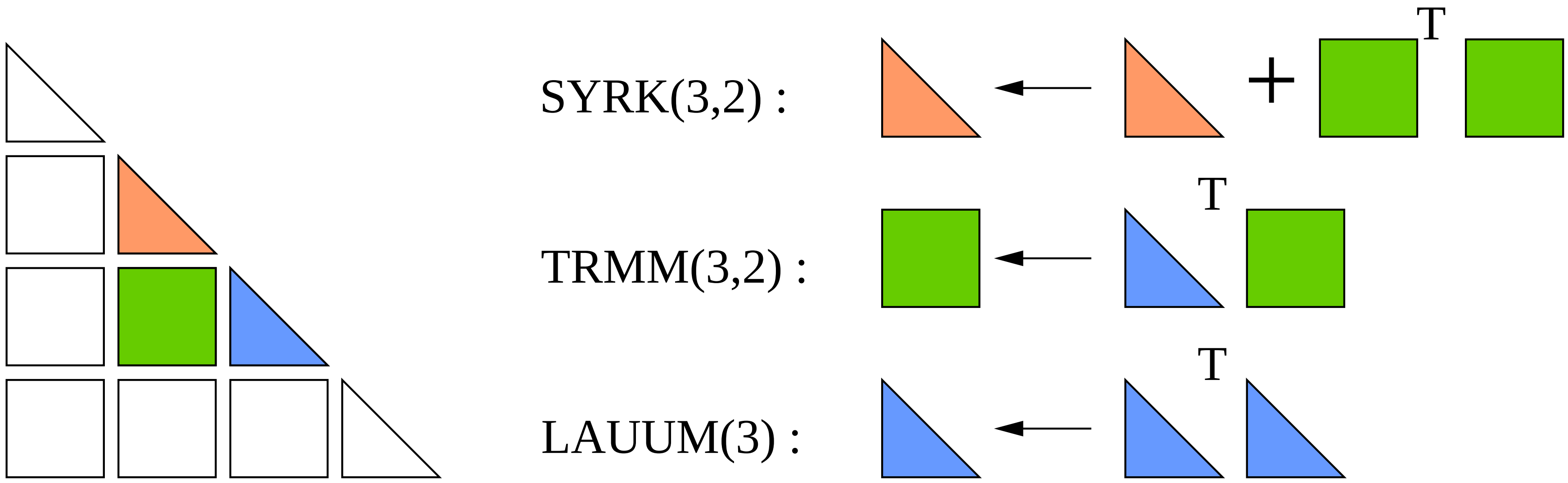
[2]: there are some assumptions pending ...

Array renaming (removing anti-dependencies)

Many of the dependencies within the DAG are a result of needing to preserve the data to allow other tasks to complete before the data can be overwritten.

For example, within LAUUM (Step 3), we have the following dependency:

SYRK(3,2) → TRMM(3,2) → LAUUM(3)



By providing a copy of the green and blue tiles, we can execute each of these tasks at the same time.

Cholesky inversion, DAGs, critical path, with some scheduling

Array renaming (removing anti-dependencies)

By using a temporary working array, we can remove these dependencies such that many tasks can run at the same time (this technique is called array renaming in compilation). In comparison to our previous algorithm, we call this out-of-place.

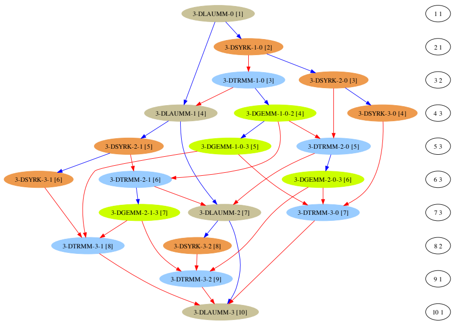
This technique provides a much higher scalability.

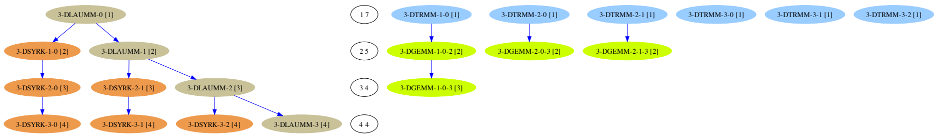
Array renaming (removing anti-dependencies)

By using a temporary working array, we can remove these dependencies such that many tasks can run at the same time (this technique is called array renaming in compilation). In comparison to our previous algorithm, we call this out-of-place.

This technique provides a much higher scalability.

Comparing the DAGs for LAUUM…





in-place : $3t-2$

out-of-place : t

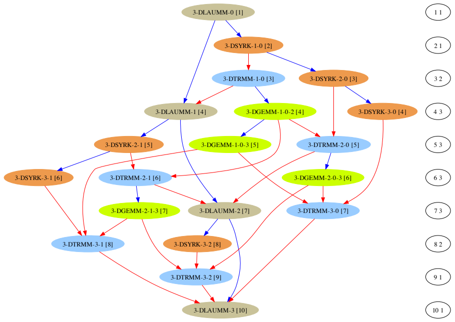
Cholesky inversion, DAGs, critical path, with some scheduling

Array renaming (removing anti-dependencies)

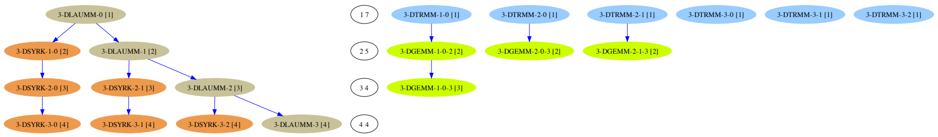
By using a temporary working array, we can remove these dependencies such that many tasks can run at the same time (this technique is called array renaming in compilation). In comparison to our previous algorithm, we call this out-of-place.

This technique provides a much higher scalability.

Comparing the DAGs for LAUUM. . .



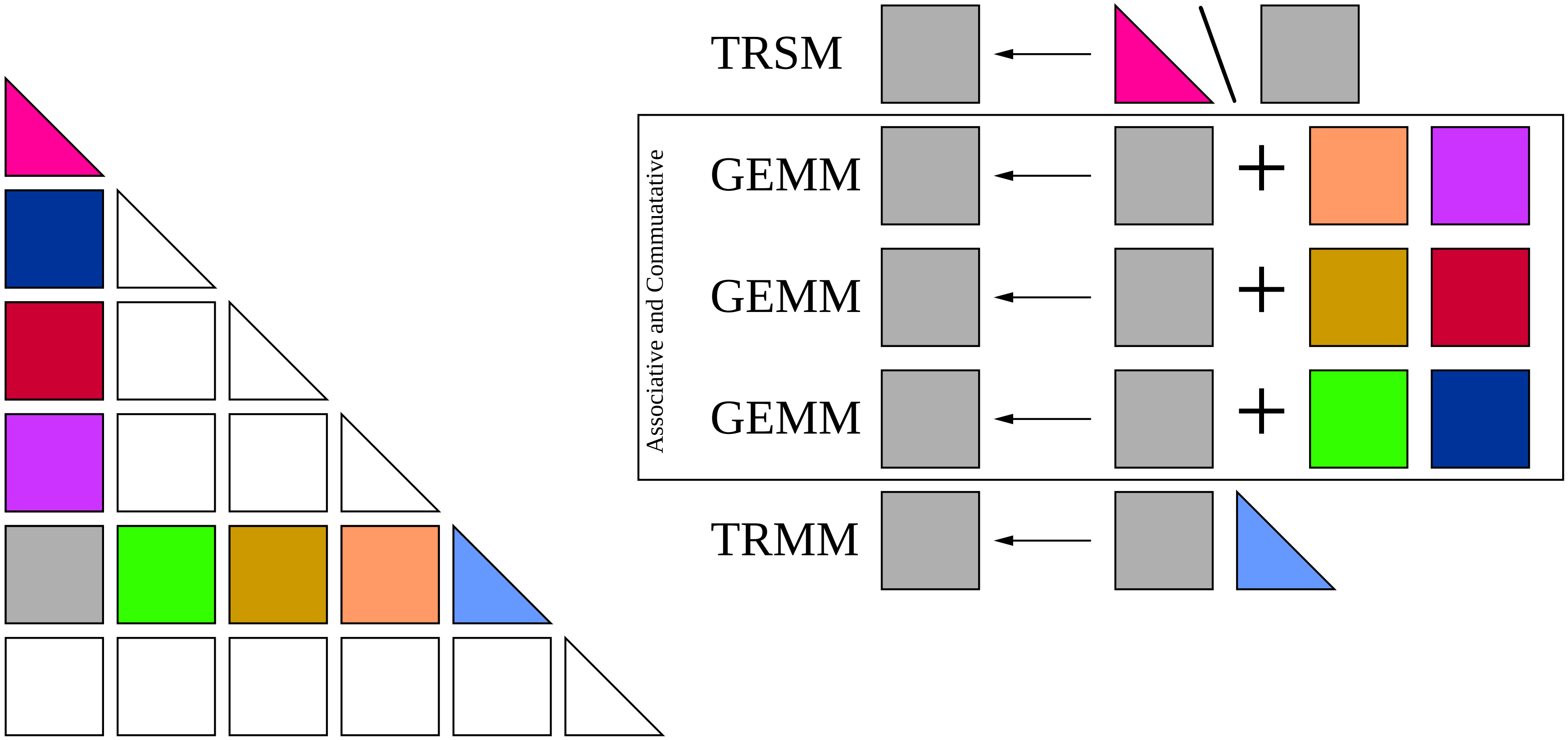
in-place : $3t - 2$



out-of-place : t

Although it would seem that the combination of pipelining with array renaming should give us better performance, from our experimental results, this was not always the case.

Loop reversal.



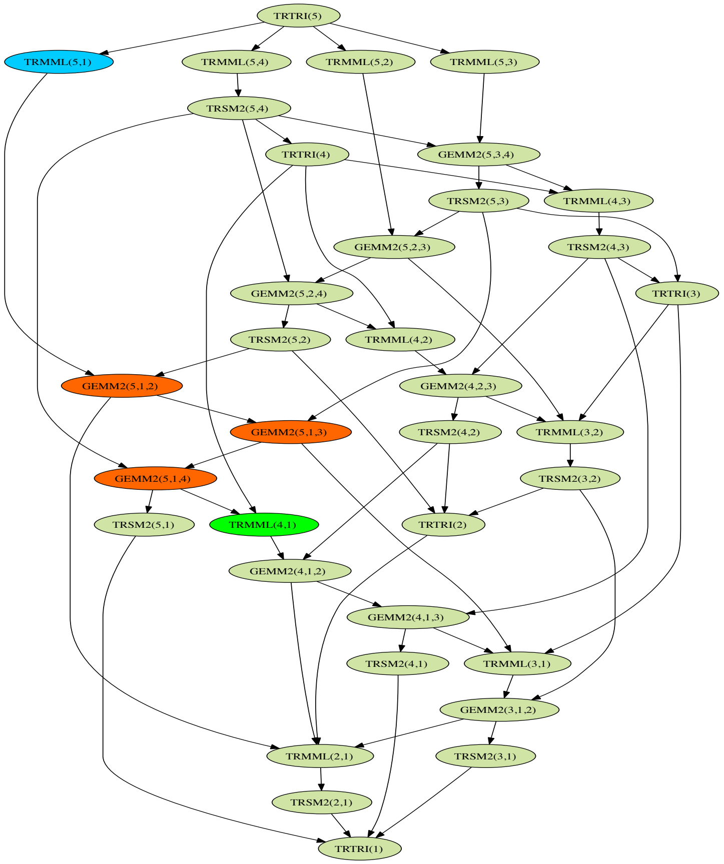
The GEMM operations are commutative, meaning we can execute them in any order in the same fashion as a DOANY loop.
We have represented the good loop here for TRTRIv4 since the saumon and purple tiles would be available way before the green and blue tiles.

Cholesky inversion, DAGs, critical path, with some scheduling

Loop reversal.

Here is TRTRIv4 with $t = 5$ where the loop ordering has not been optimized to take advantage of the commutativity. (The loop goes UP.)

```
Step 2: Tile Triangular Inversion of $L$
(compute $L^{-1}$);
for $j = t-1$ to $0$ do
    for $i = t-1$ to $j+1$ do
        $A_{i,j} \leftarrow A_{i,i} * A_{i,j}$ (TRMM(i,j)) ;
        for $k = j+1$ to $i-1$ do
            $A_{i,j} \leftarrow A_{i,j} + A_{i,k} * A_{k,j}$
            (GEMM(i,j,k)) ;
        end
    end
    for $i = j+1$ to $t-1$ do
        $A_{i,j} \leftarrow -A_{i,j} / A_{j,j}$ (TRSM(i,j)) ;
    end
    $A_{j,j} \leftarrow TRINV(A_{j,j})$ (TRTRI(j)) ;
end
```



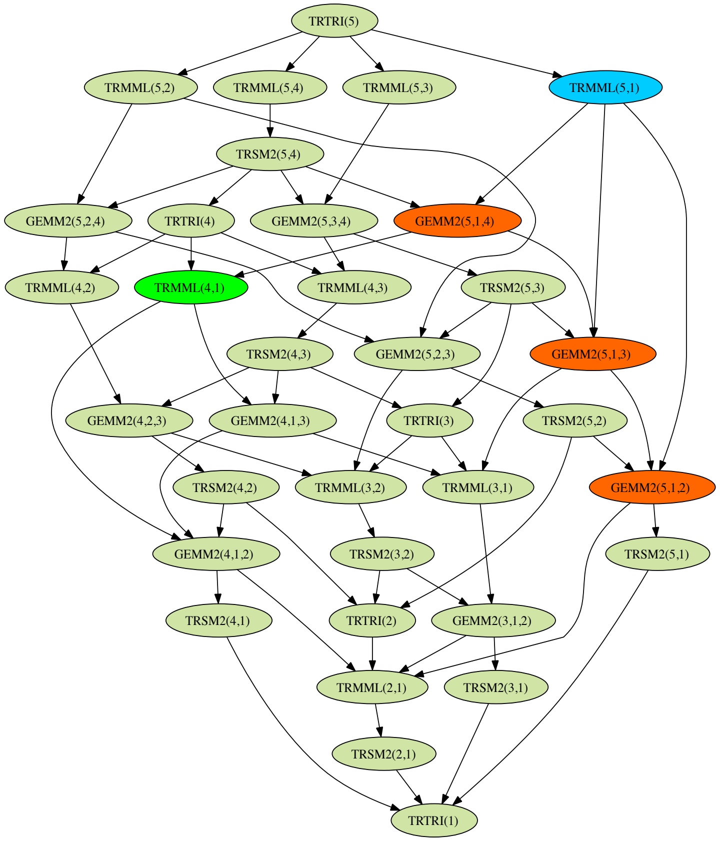
Critical path : $19 = t^2 - t - 1$

Cholesky inversion, DAGs, critical path, with some scheduling

Loop reversal.

The same algorithm (TRTRIv4, $t = 5$), but the indexing on the loops has been altered to fully take advantage of the commutativity. (The loop goes DOWN.)

```
Step 2: Tile Triangular Inversion of $L$ (compute $L^{-1}$);
for $j = t-1$ to $0$ do
    for $i = t-1$ to $j+1$ do
        $A_{i,j} \leftarrow A_{i,i} * A_{i,j}$ (TRMM(i,j)) ;
        for $k = i-1$ to $j+1$ do
            $A_{i,j} \leftarrow A_{i,j} + A_{i,k} * A_{k,j}$ (GEMM(i,j,k)) ;
        end
    end
    for $i = j+1$ to $t-1$ do
        $A_{i,j} \leftarrow -A_{i,j}/A_{j,j}$ (TRSM(i,j)) ;
    end
    $A_{j,j} \leftarrow TRINV(A_{j,j})$ (TRTRI(j)) ;
end
```

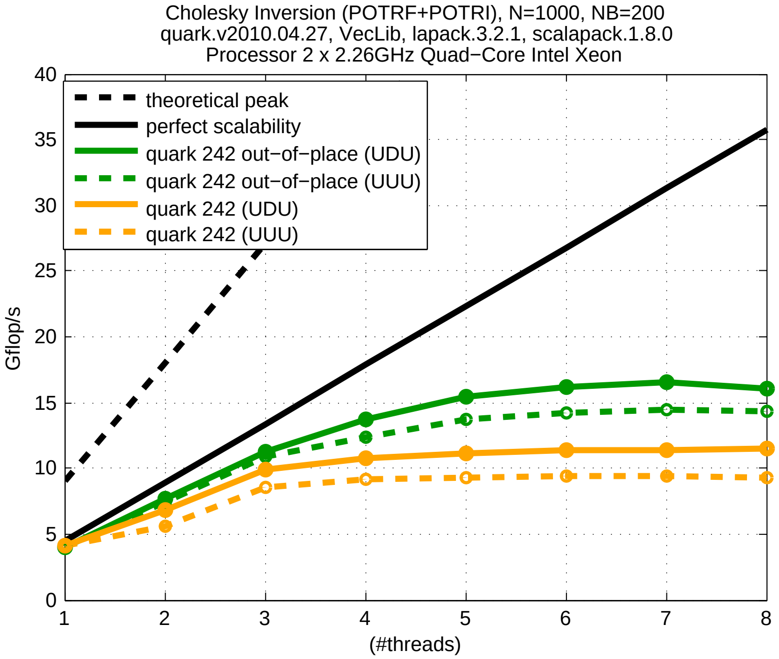


Critical path : 13 = $3t - 2$

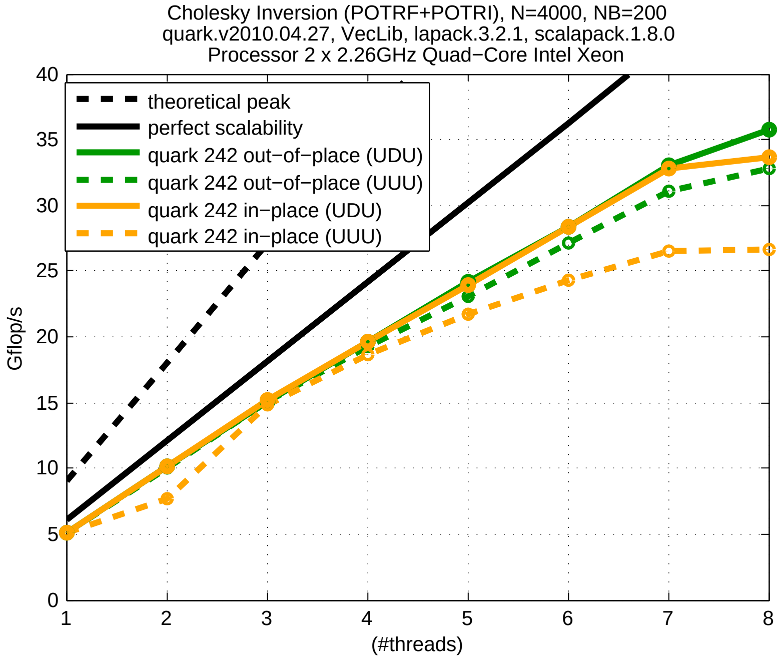
Cholesky inversion, DAGs, critical path, with some scheduling

Experimental results
We do not have tuned the scheduler (e.g. LOCALITY flag), no hint.

Using a dynamic scheduler, we compared the effects of loop reversal and array renaming.



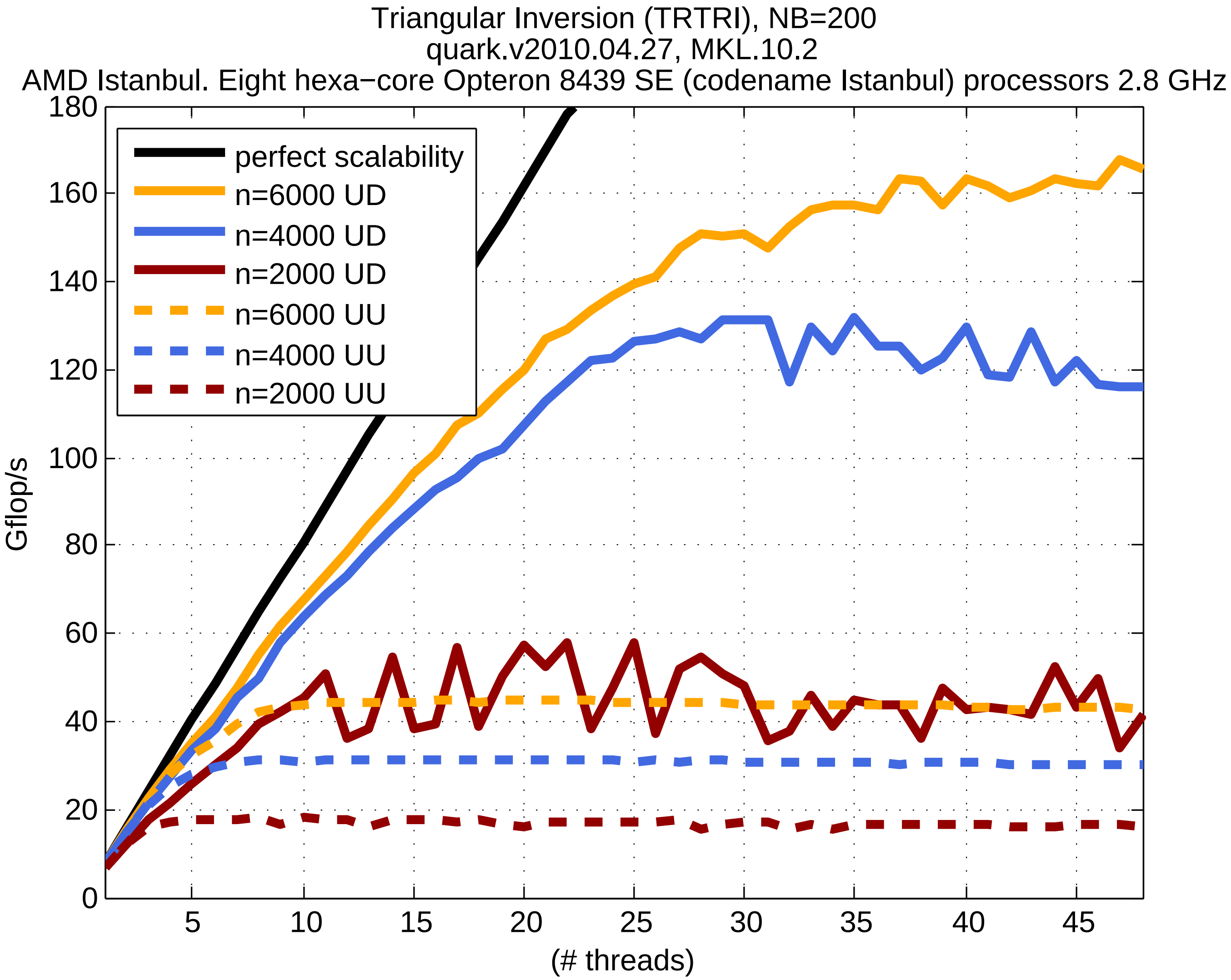
$n = 1000$



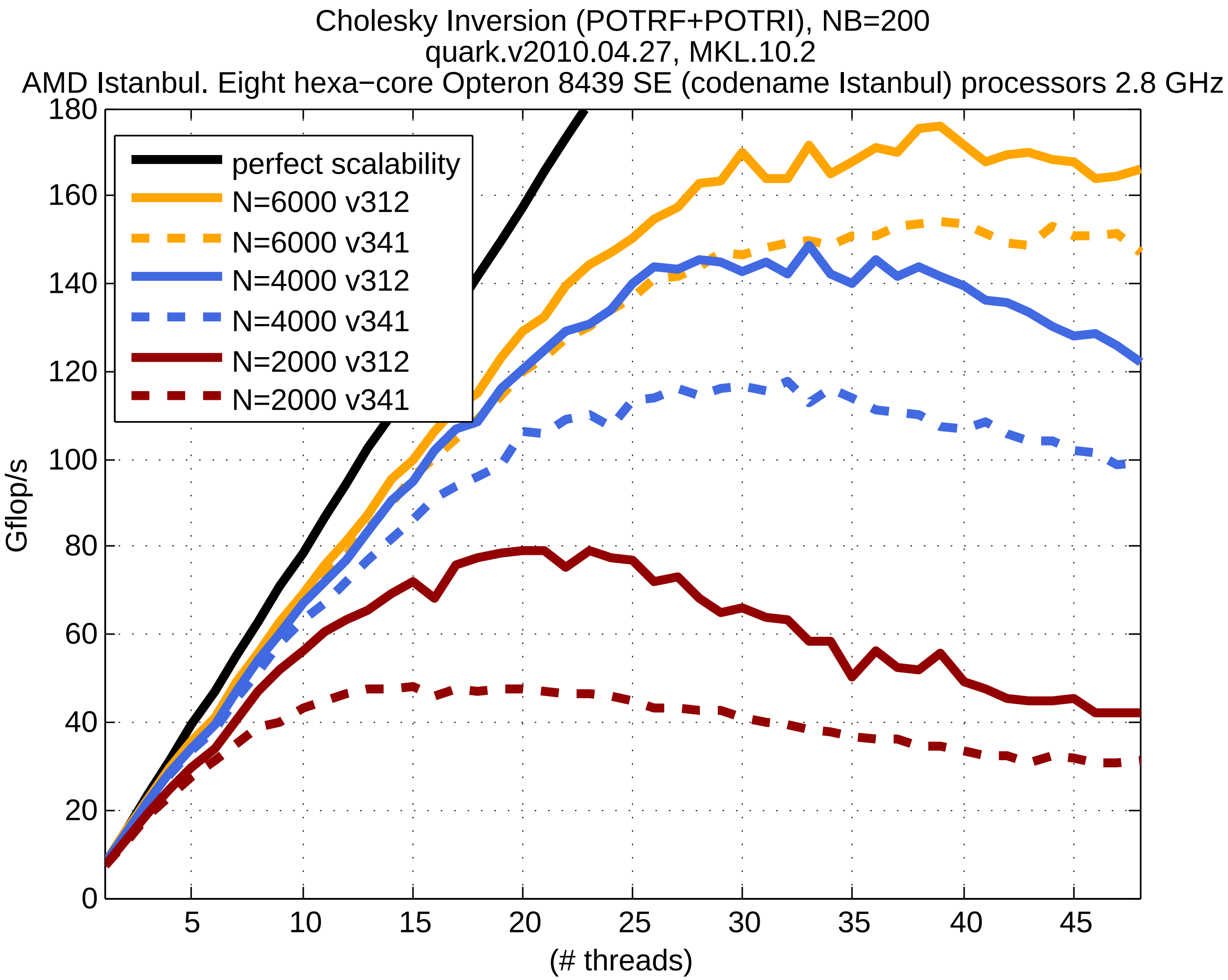
$n = 4000$

Note: on NUMA architecture, we did not manage to get speed up with array renaming. (More precisely, we managed to get great slowdown ...)

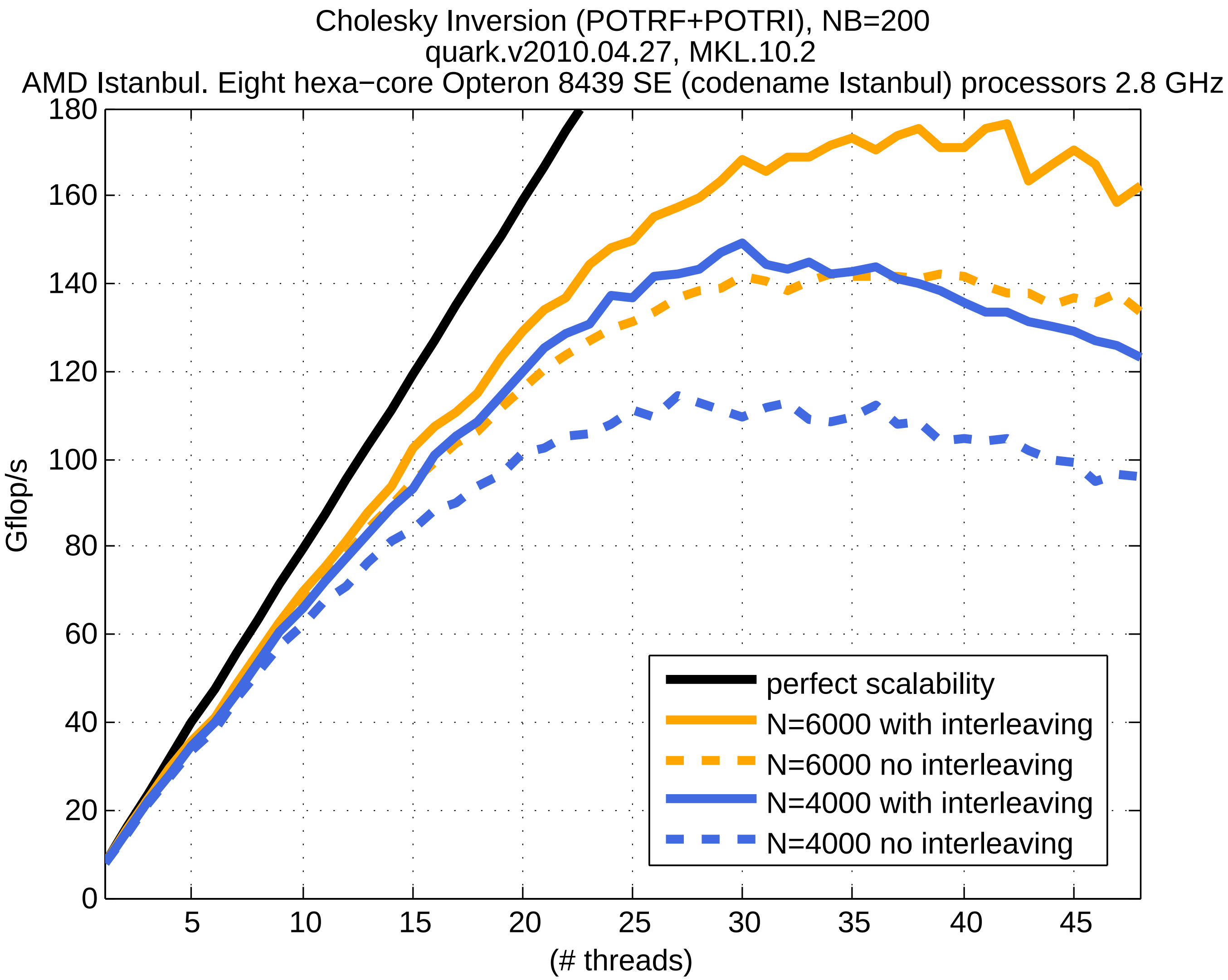
TRTRIv4: Comparison of variant UU $(t^2 - t - 1)$ and variant UD $(3t - 2)$



CHOLINV: Comparison of variant 312 $(3t+2)$ and variant 341 $(9t-6)$



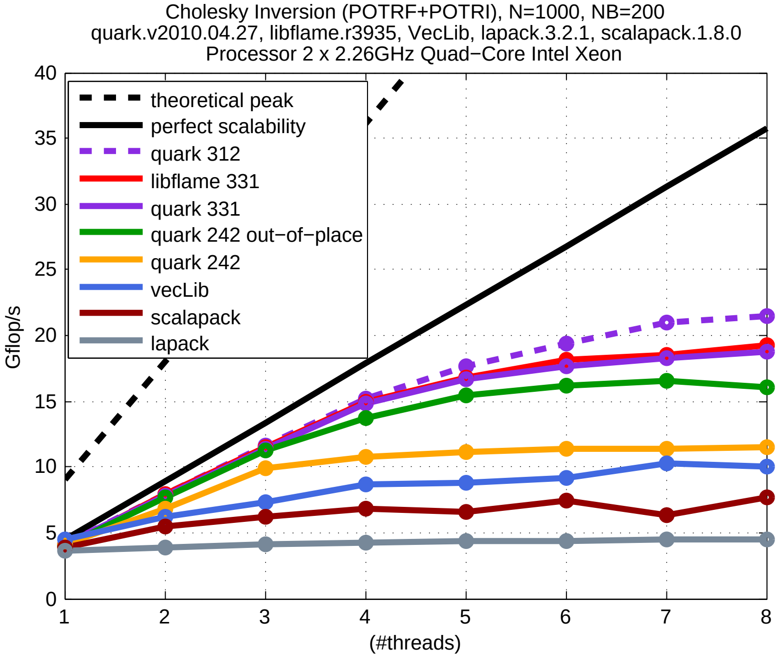
CHOLINV 312: Comparison of with interleaving and no interleaving



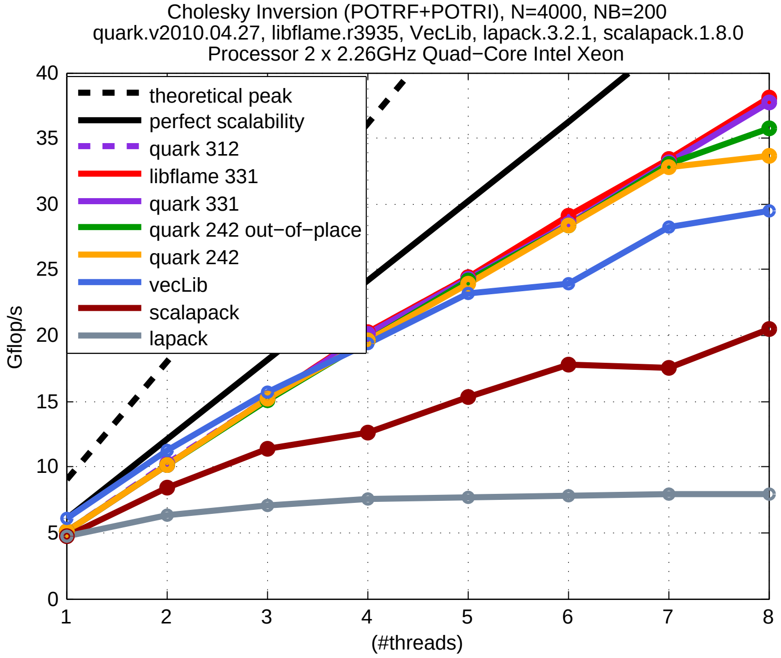
Cholesky inversion, DAGs, critical path, with some scheduling

Experimental results

Using a dynamic scheduler, our experiments consistently achieved significantly better performance than vecLib, ScaLAPACK and LAPACK libraries.

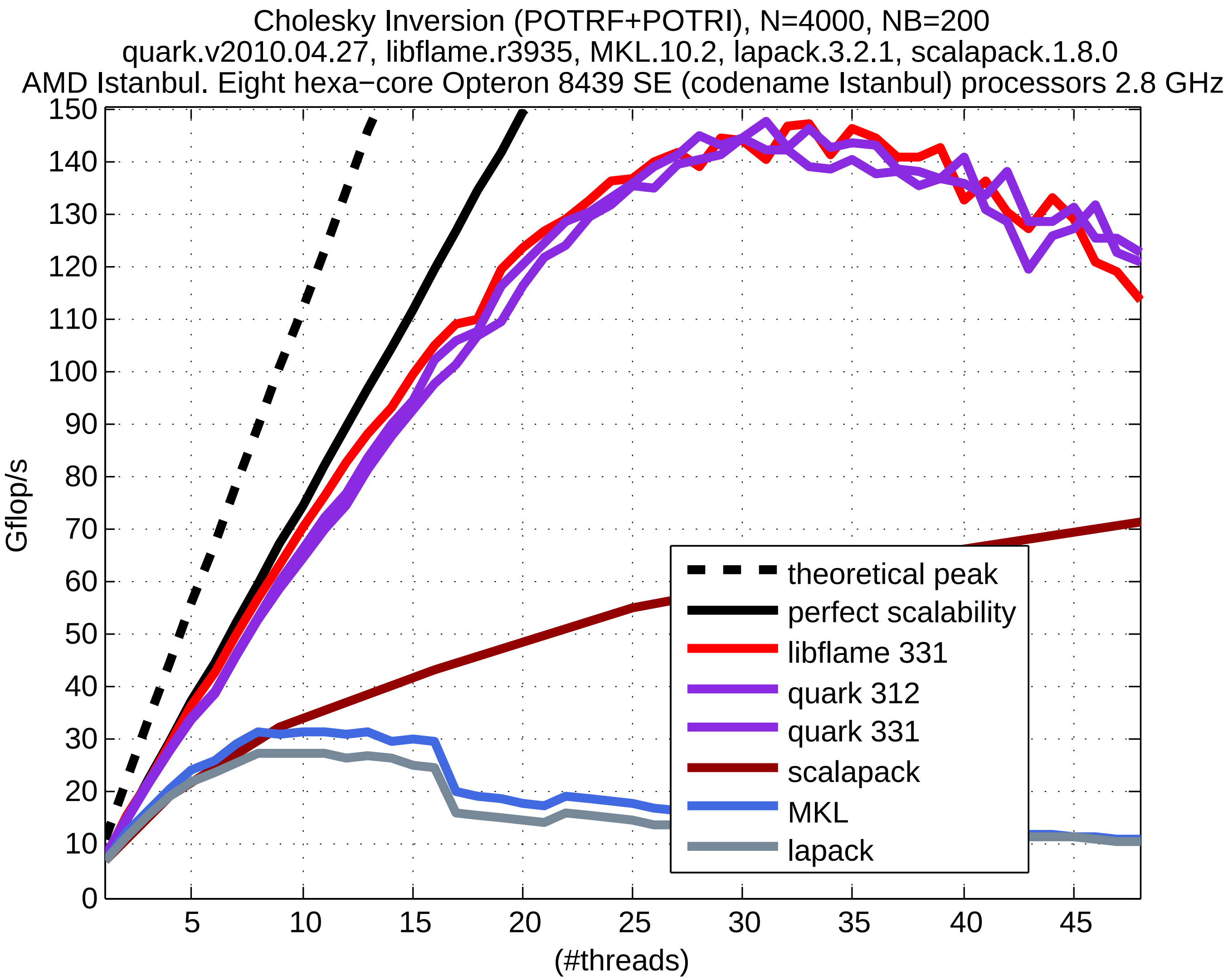


$n = 1000$

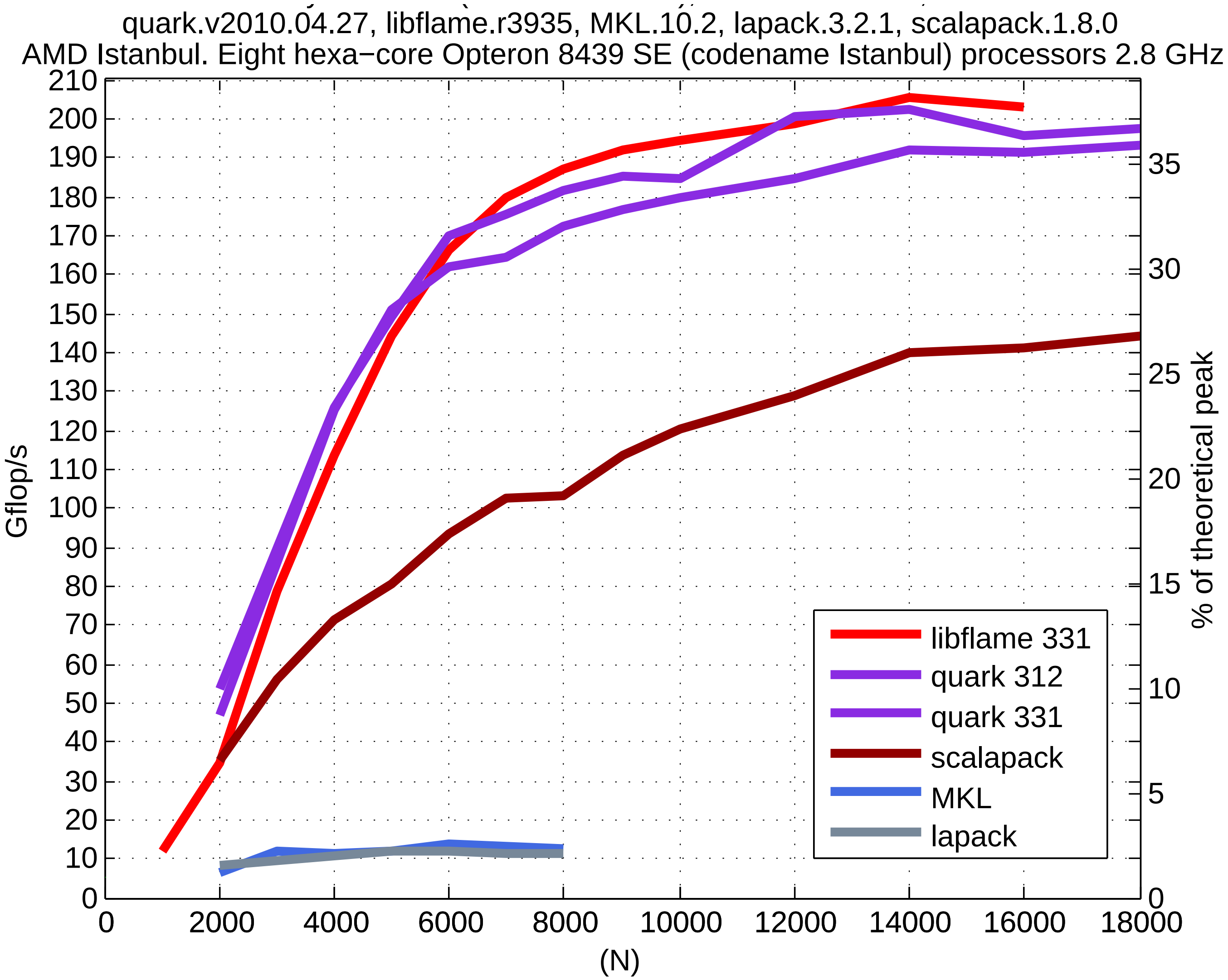


$n = 4000$

CHOLINV: performance results



CHOLINV: performance results



Note: the top of the y-axis is about 40% of the peak performance.

Cholesky inversion, DAGs, critical path, with some scheduling

Mission accomplished?

Criteria for selection. We keep the granularity of our algorithms constant (the block size), then an algorithm is said to be better than another if it has a shorter critical path. We count the length in term of tasks (as opposed to flops).

Some thoughts: Is our criterion relevant after all?

1. Same analysis with flops!
2. The critical path is nice because we can compute it but this is not the whole story in term of parallelism.
3. We are often bounded by the bandwidth as opposed to the parallelism. (Need to model caches.)
4. But yes! Our criterion is a start and we do understand experimental results better thanks to it.

The main messages.

1. We are able at the algorithm level to "change" the DAG of an application.[3]
2. Existing compiler techniques can be used in our work. Since the granularity of our tasks is "huge" (relatively to a few flops), compilers/schedulers have many more opportunities to "think" about the right thing to do.

Other research in that vein.

We consider the QR factorization of (tall and skinny) matrix as a reduction. We can then play with the reduction tree to optimize for the underlying architecture, or what needs to be done after with the tree (apply Q^T, compute Q, nothing.)

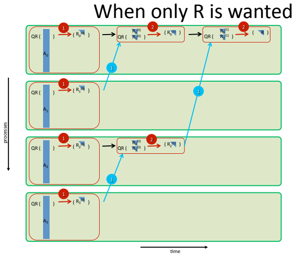
Our goal was to provide a good DAG

and to leave the rest as a scheduling problem!

Cholesky inversion, DAGs, critical path, with some scheduling

Implementing Communication-Optimal Parallel and Sequential QR Factorizations

When only R is wanted



Implementing Communication-Optimal Parallel and Sequential QR Factorizations

When only R is wanted: The MPI_Allreduce

In the case where only R is wanted, instead of constructing our own tree, one can simply use MPI_Allreduce with a user defined operation. The operation we give to MPI is basically the Algorithm 2. It performs the operation:

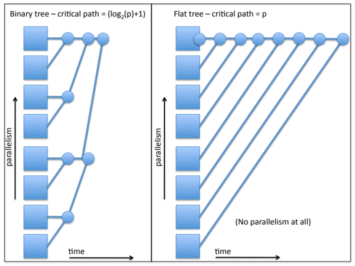
$$\mathrm{QR}\left(\begin{array}{c} R_1 \\ R_2 \end{array}\right) \longrightarrow R$$

This **binary** operation is **associative** and this is all MPI needs to use a user-defined operation on a user-defined datatype. Moreover, if we change the signs of the elements of R so that the diagonal of R holds positive elements then the binary operation `Rfactor` becomes **commutative**.

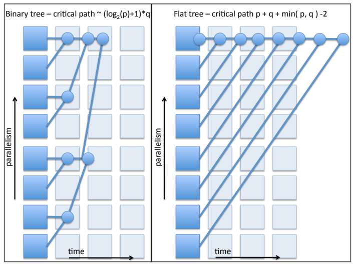
The code becomes two lines:

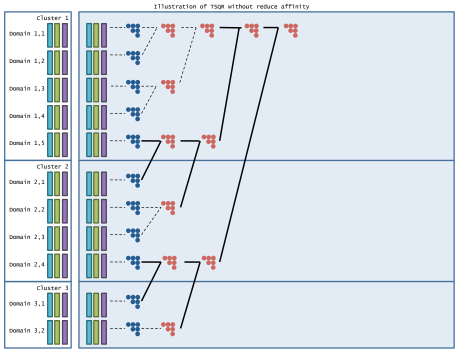
```
lapack_dgeqrf( mloc, n, A, lda, tau, &dlwork, lwork, &info );
MPI_Allreduce( MPI_IN_PLACE, A, 1, MPI_UPPER,
        LILA_MPIOP_QR_UPPER, mpi_comm);
```

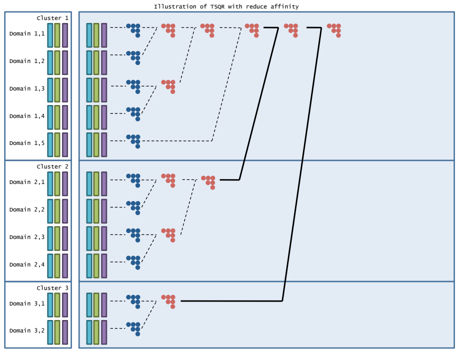
Implementing Communication-Optimal Parallel and Sequential QR Factorizations

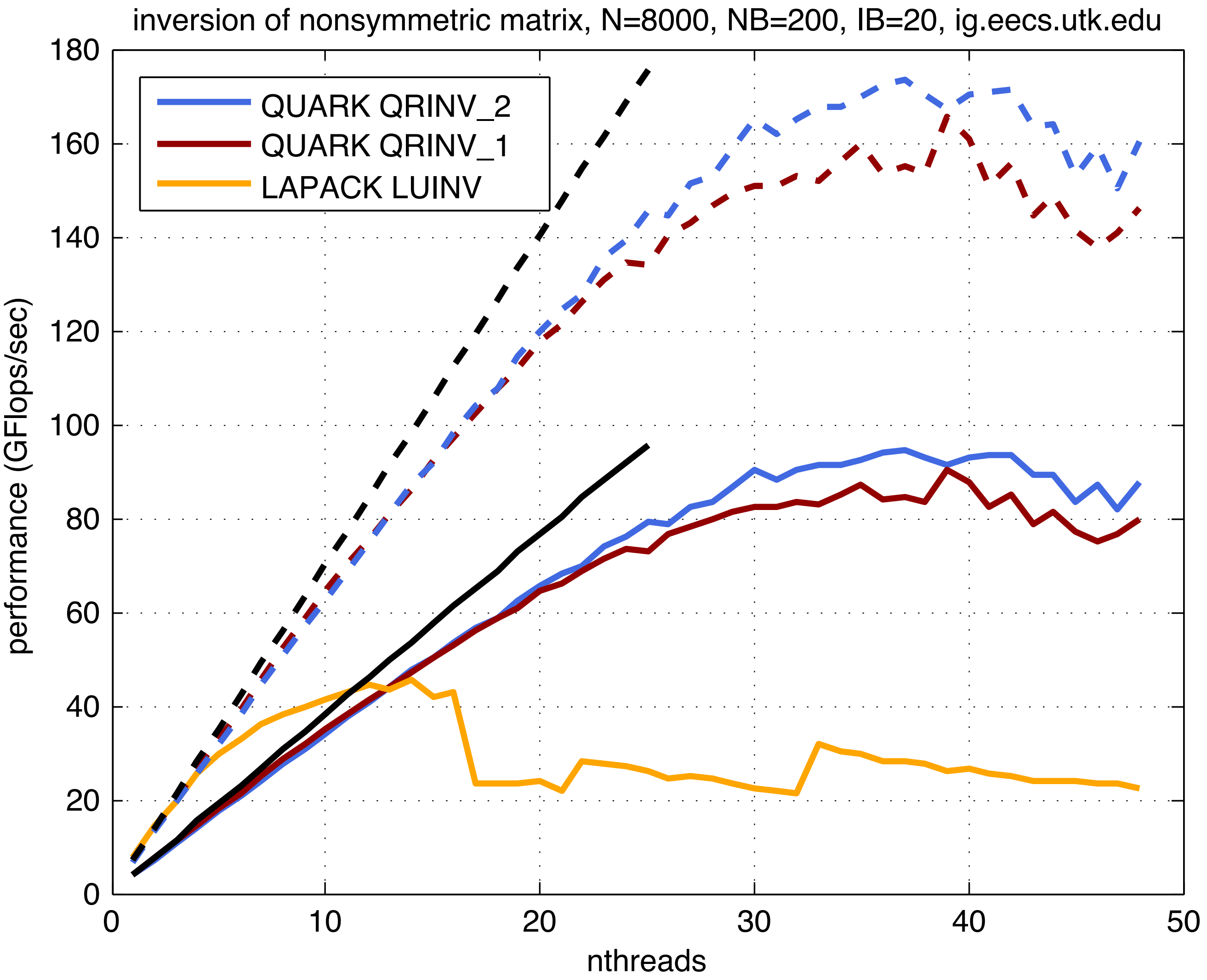

Binary tree – critical path = (log2(p)+1)
Flat tree – critical path = p
parallelism
parallelism
(No parallelism at all)
time
time

Implementing Communication-Optimal Parallel and Sequential QR Factorizations


Binary tree – critical path ~ (log2(p)+1)*q
Flat tree – critical path p + q + min(p, q) -2
parallelism
parallelism
time
time

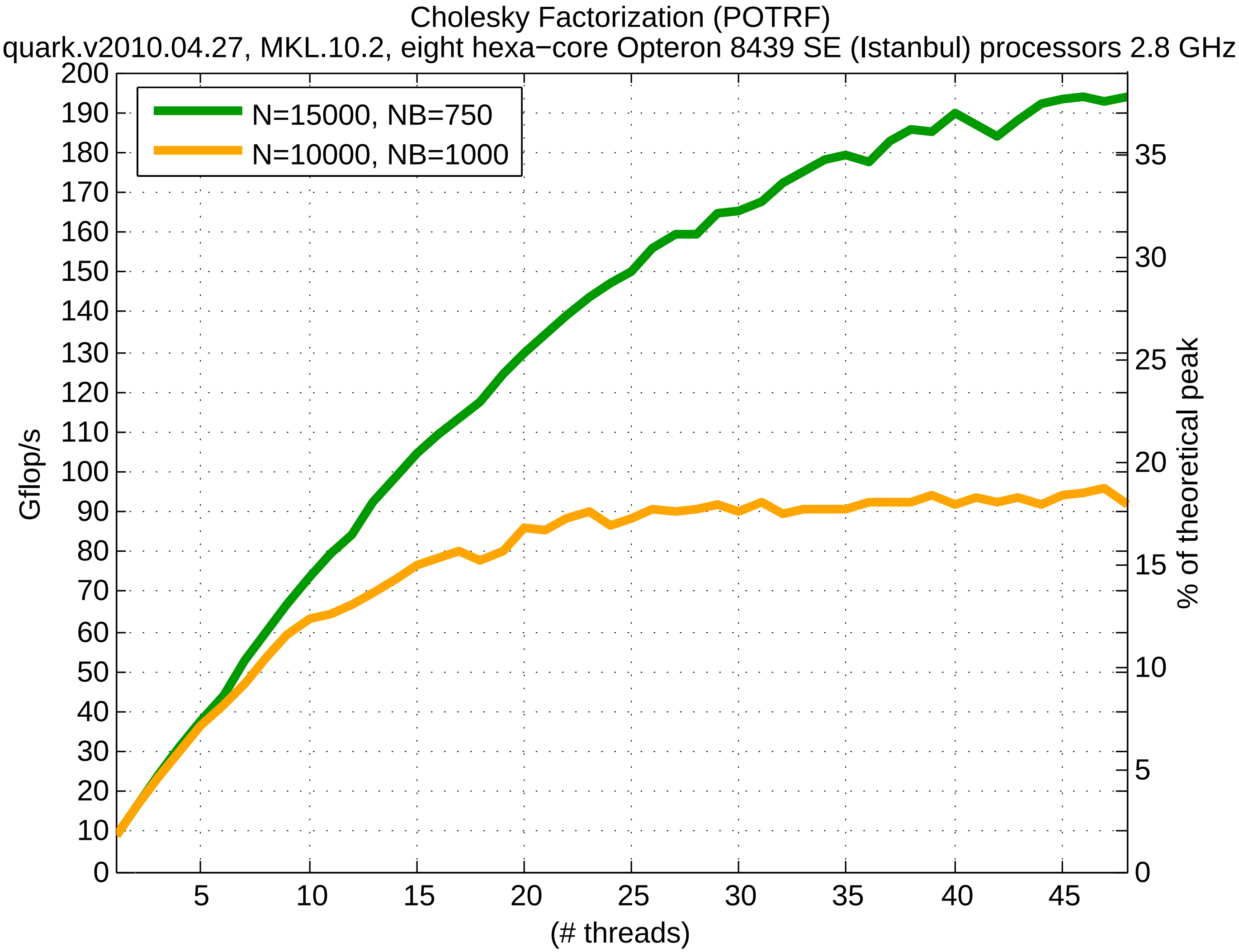





inversion of nonsymmetric matrix, N=8000, NB=200, IB=20, ig.eecs.utk.edu
QUARK QRINV_2
QUARK QRINV_1
LAPACK LUINV
performance (GFlops/sec)
nthreads
0
20
40
60
80
100
120
140
160
180
0
10
20
30
40
50

New mission!

Let us look at:



Cholesky Factorization (POTRF)
quark.v2010.04.27, MKL.10.2, eight hexa−core Opteron 8439 SE (Istanbul) processors 2.8 GHz

Note: we have $t = 20$ and $t = 10$ (big) tiles.

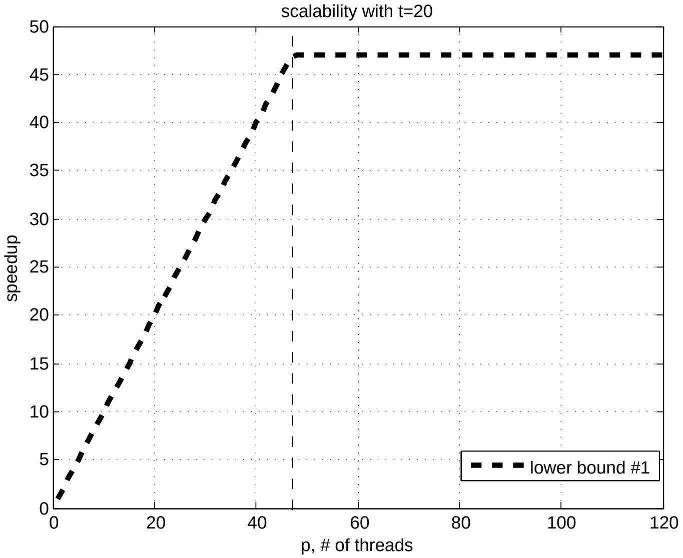
We consider a n–by–n matrix with nb–by–nb tiles, and set $n = t * nb$.
We take for unit of flops $nb^3/3$.
This makes n and nb disappears from our problem.

The number total of flops in Cholesky is t^3 $(nb^3/3)$.
The length of the critical path is $9t - 10$ $(nb^3/3)$.

Therefore on p threads, the execution time of Cholesky is at least

$$\max(\frac{t^3}{p}, 9t - 10).$$

This is our first lower bound for any execution time. (So in particular for the optimal execution time.)



Note: the expected speedup is therefore less than this dashed line. (The lower bound in time becomes a upper bound in speedup.)

How well does this lower bound perform?

Let us try some scheduling heuristics!

We consider three strategies:

1. **max** list schedule with priority given to the task with the maximum flops in all of its children (and itself),
2. **rand** list schedule with random tasks selection
3. **min** list schedule with priority given to the task with the minimum flops in all of its children (and itself),

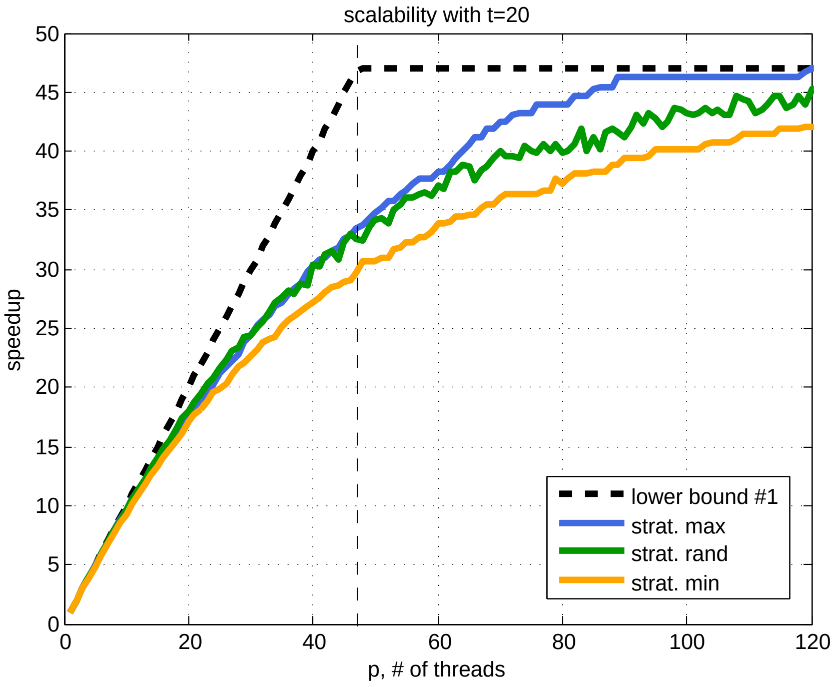
Obviously, strategy min is not a good idea! We hope it gives a sense of what the worse list schedule can be. Note for the max and min strategies, we have the (convenient) explicit formulae

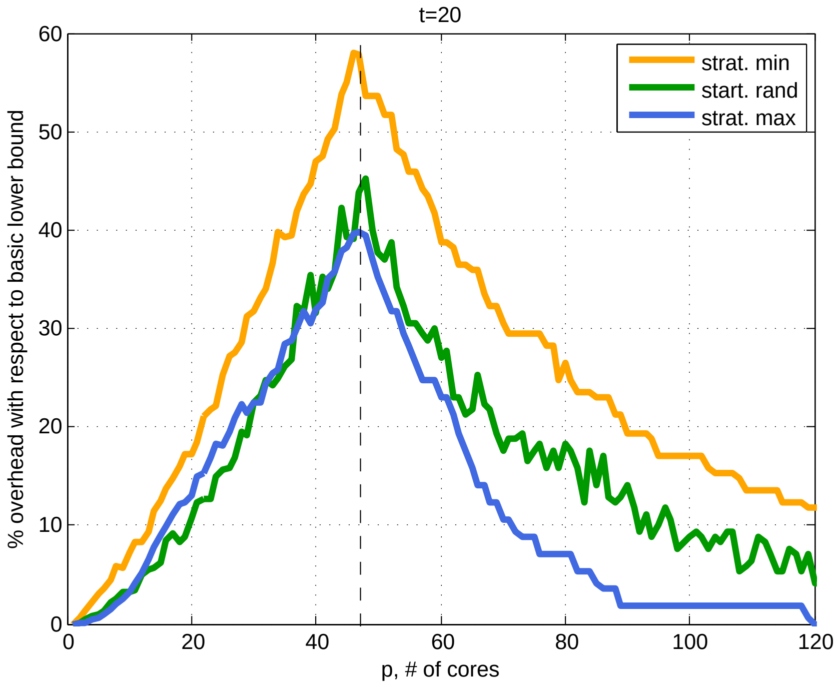
$$\begin{aligned}
\texttt{POTRF\_i\_\_\_priority}(i,t) &= (t-i)^3, \\
\texttt{GEMM\_\_ijk\_priority}(i,j,k,t) &= (t-j)^3+3(j-i)(2t-i-j-3)+6(j-k), \\
\texttt{SYRK\_\_ij\_\_priority}(i,j,t) &= (t-i)^3+3(i-j), \\
\texttt{TRSM\_\_ij\_\_priority}(i,j,t) &= (t-j)^3+3(j-i)(2t-i-j-1).
\end{aligned}$$

Cholesky inversion, DAGs, critical path, with some scheduling

We consider three strategies:

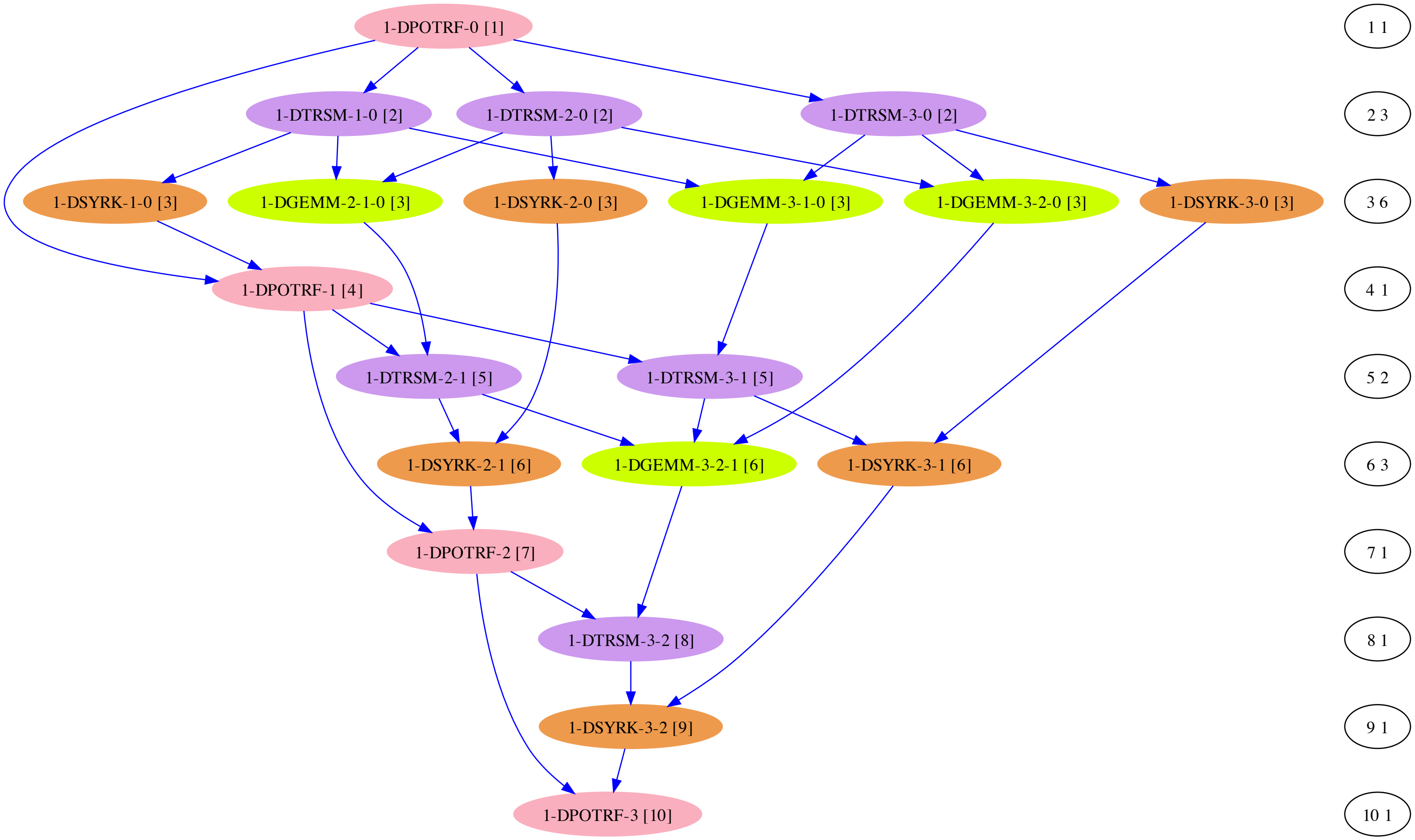
1. **max** list schedule with priority given to the task with the maximum flops in all of its children (and itself),
2. **rand** list schedule with random tasks selection
3. **min** list schedule with priority given to the task with the minimum flops in all of its children (and itself),





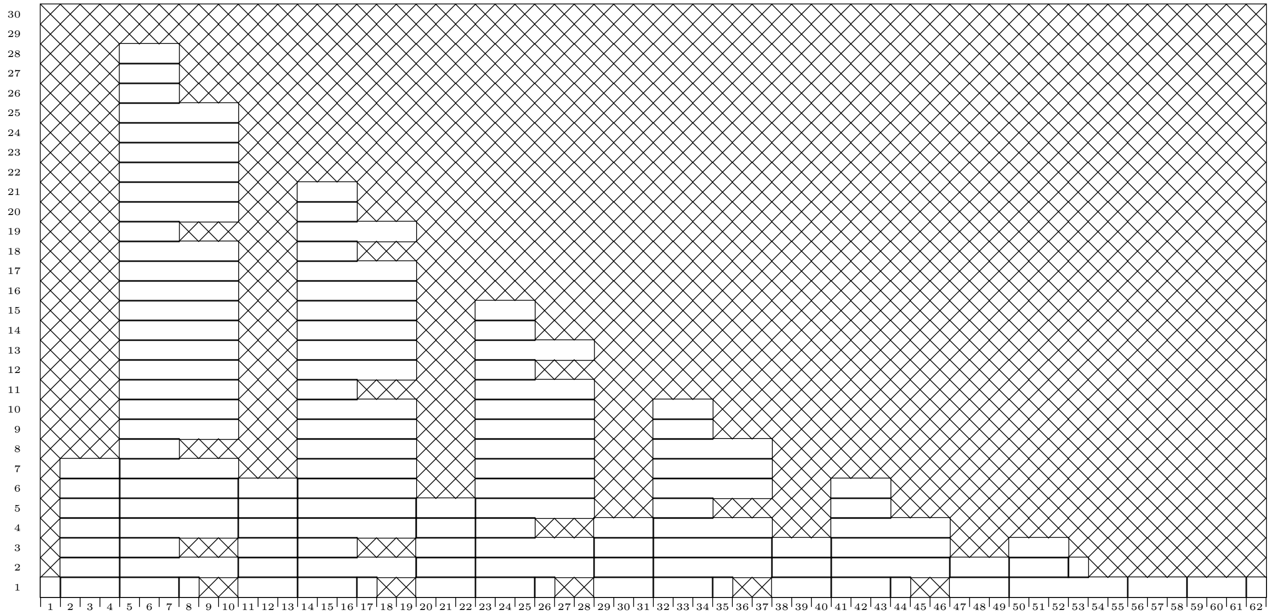
If we are not happy with this 40% discrepancy between the lower bound and the upper bound, we have two choices ... either find a greater lower bound or find a lower upper bound ... or both ...

Cholesky inversion, DAGs, critical path, with some scheduling

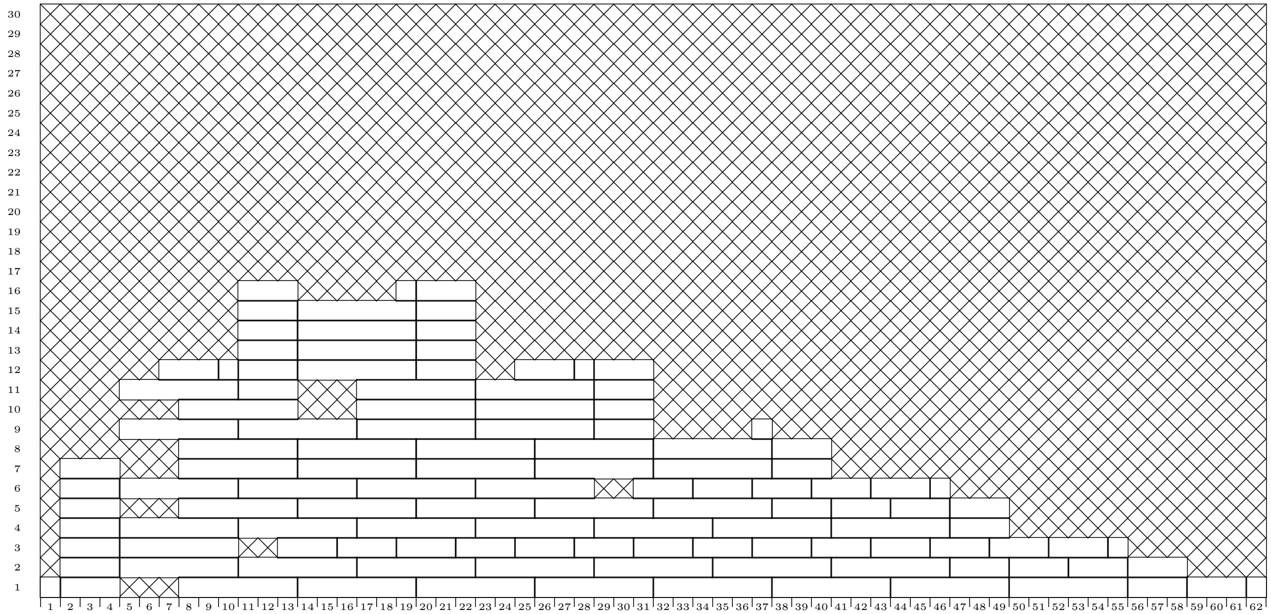


Cholesky inversion, DAGs, critical path, with some scheduling

Forward list schedule $t = 8$ and $p = 30$.



Reverse "list schedule" $t = 8$ and $p = 30$.



$t = 8$
Length of the critical path: 82

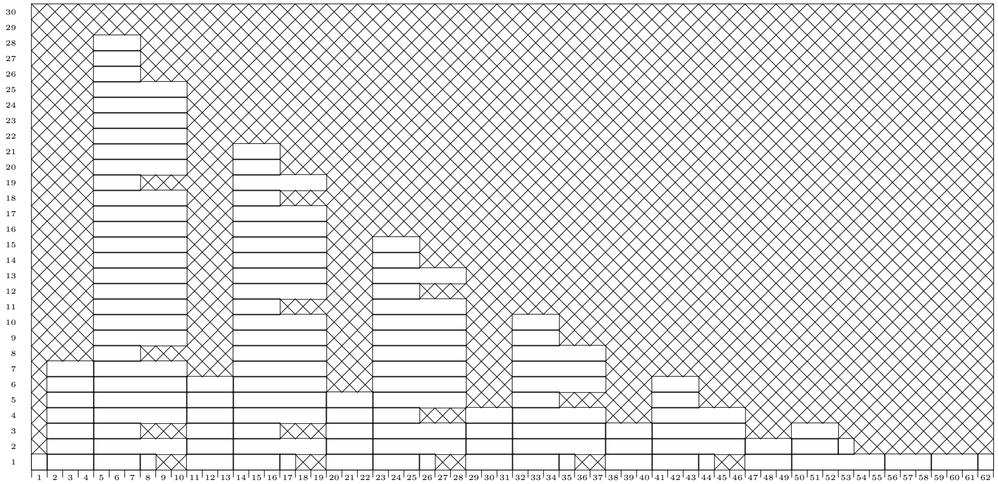
We compare two strategies: list schedule forward and list schedule reverse.

Since there is too many processors (30) in either case (forward and backward), the list schedule does not need a strategy to go through.

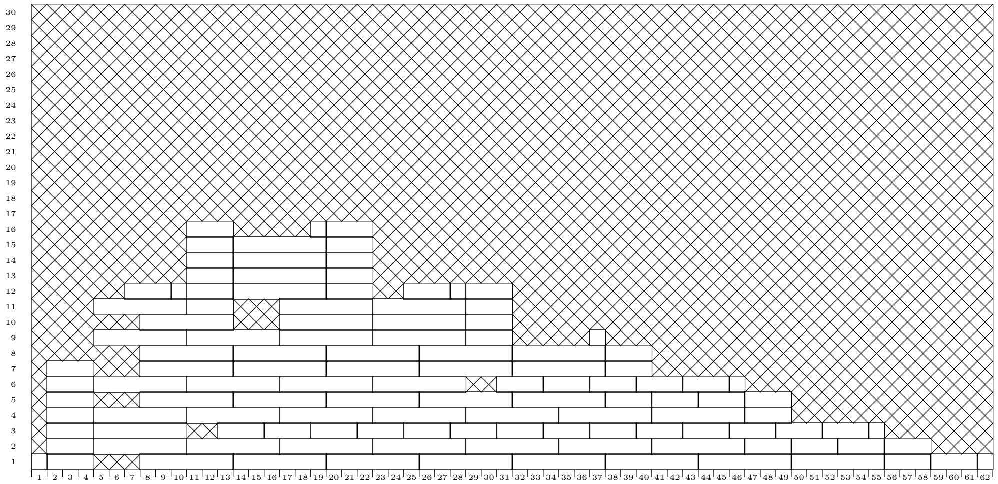
We see that forward "needs" more processors than reverse.

Note the reverse "list schedule" is not a list schedule ...

Forward list schedule $t = 8$ and $p = 30$



Reverse “list schedule” $t = 8$ and $p = 30$.

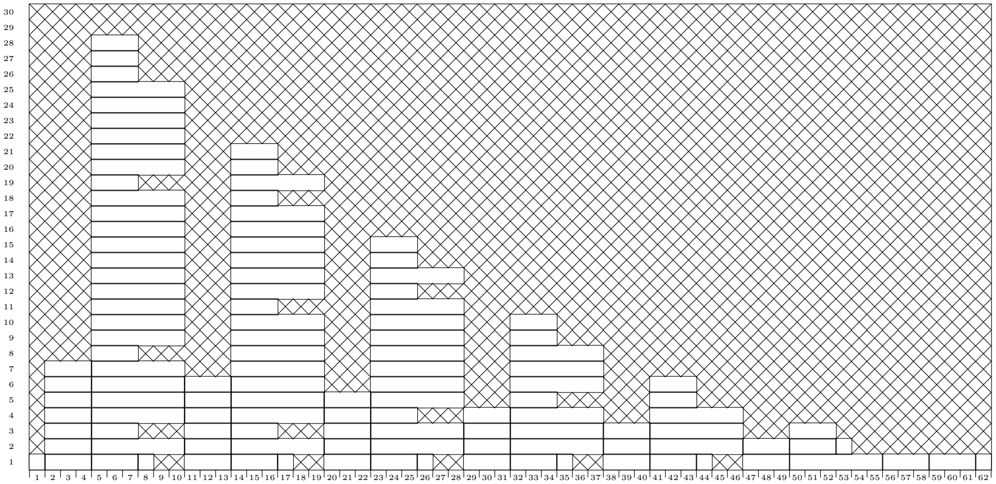


If a task k starts in time step j of the forward simulation, this implies that it cannot be started any sooner than time step j. (For any number of processors, for any scheduling strategies.)

Conversely, if a task k ends in time step j of the reverse simulation, this implies that it cannot be ended any sooner than time step `endofrun` $- j$. (For any number of processors, for any scheduling strategies.)

In some sense the forward simulation represents the best parallelism that you can have at the start, the reverse simulation represents the best parallelism that you can have at the end.

Forward list schedule $t = 8$ and $p = 30$



If a task k starts in time step j of the forward simulation, this implies that it cannot be started any sooner than time step j. (For any number of processors, for any scheduling strategies.)

Therefore, if

$$\forall i = 1, \ldots, j, \quad w_F(i) < p,$$

we can increase our lower bound

$$\text{from} \quad \frac{t^3}{p} \quad \text{to} \; \frac{t^3 - \sum_{i=1}^{k} w_F(j)}{p} + k.$$

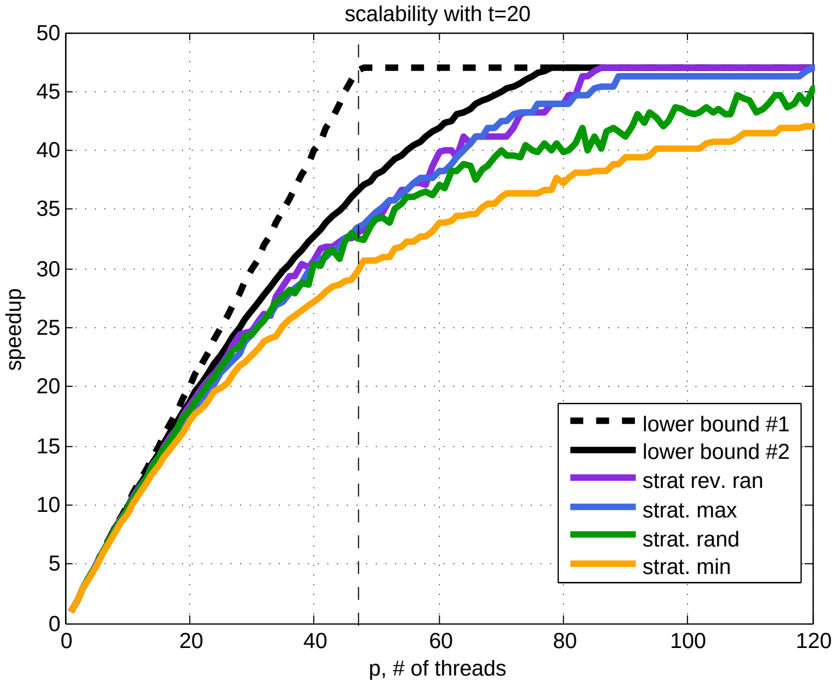
In other words, in this case, it is best (and legitimate) to count the contribution of step i as 1 as to count it as $w_F(i)/p$.

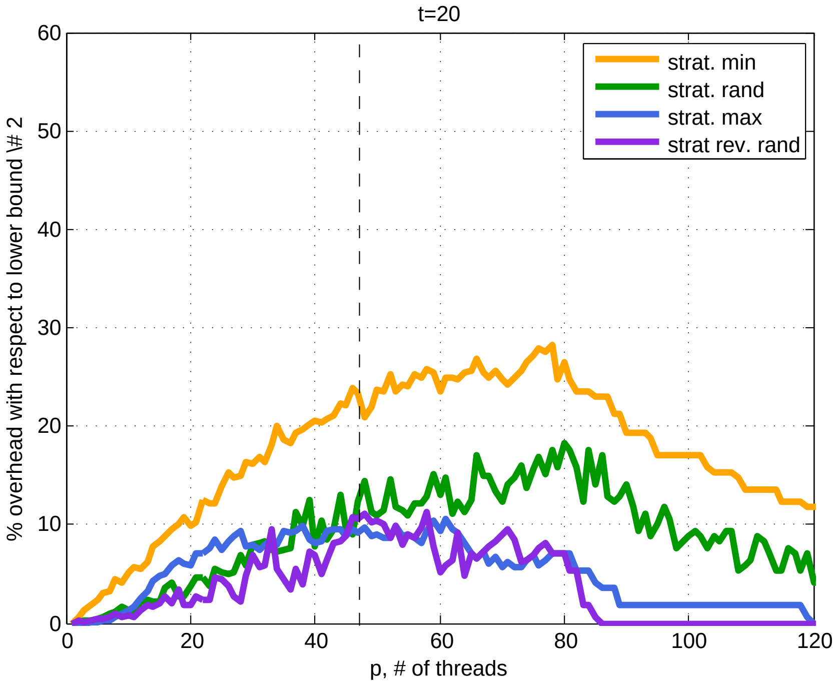
So, for example, for Cholesky, since the first step is always one task, it is best to count it in execution time as 1 as $1/p$.

The same idea applies for the reverse schedule.

Cholesky inversion, DAGs, critical path, with some scheduling

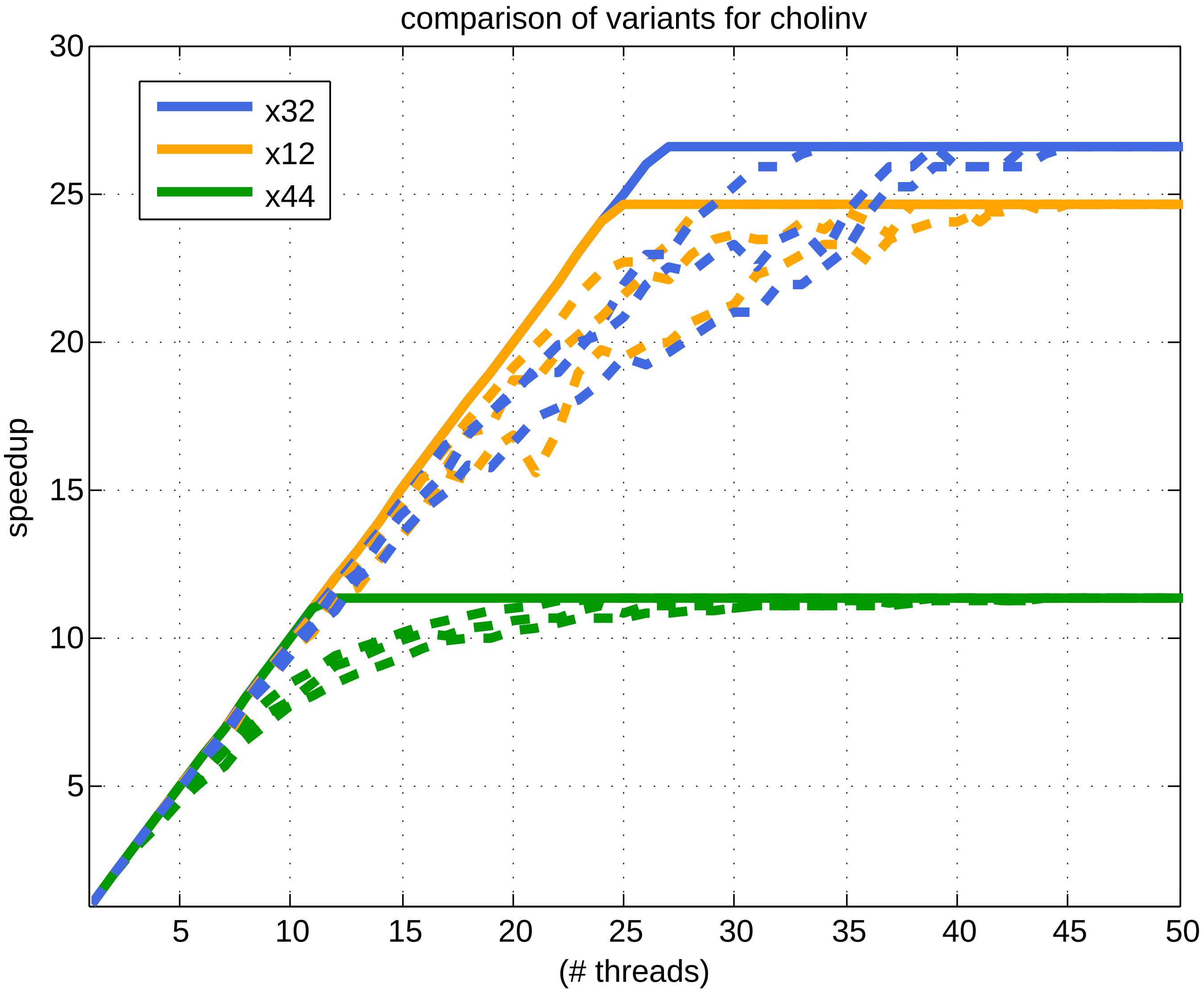
Note: rand reverse is “random reverse list schedule” strategy.





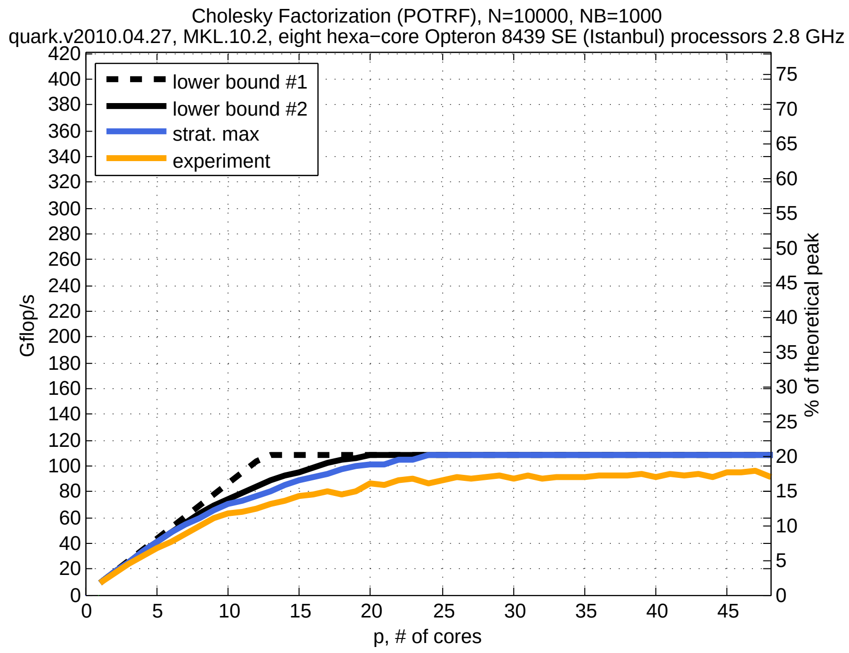
If we are not happy with this ~~40%~~ 10% discrepancy between the lower bound and the upper bound, we have two choices ... either find a greater lower bound or find a lower upper bound ... or both ...

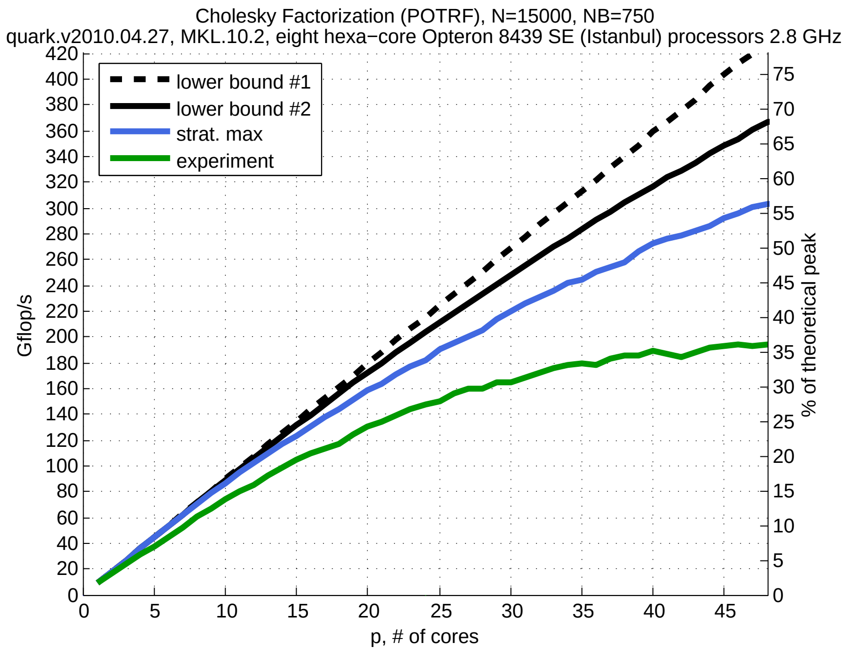
The same approach can be done for Cholesky inversion:



There is an interesting case to extend our lower bound (#2) for the case of POTRF+TRTRIv4+LAUMMv2. In this case, the DAG is a succession of three diamonds. We can extend our methodology to this case as well.

Cholesky inversion, DAGs, critical path, with some scheduling





Not quite yet there ... Even for $nb = 750$, we note that the bandwidth is not negligible. The application is actually latency bounded (as opposed to parallelism bounded).

Cholesky inversion, DAGs, critical path, with some scheduling

Related items.

1. the “reverse” idea has been used to provide a greater lower bound however this idea can also be used to obtain competitive scheduling strategy. Since the width of the representation of the DAG is smaller there is less opportunity for “mistake” by starting by the end, there is less choice, more good decision. In particular a random reverse scheduling strategy is very competitive.
2. we also have studied the problem of finding the minimum of processors in which we can realize the critical path length.
3. we have extended our lower bound approach to consecutive diamonds (useful for matrix inversion for example.)

TODO list.

1. we would like to study other scheduling strategies as FIFO Cholesky left looking, FIFO Cholesky right looking , etc. (Just in case.)
2. perform some exhaustive search for finding the optimal scheduling strategies for small cases.
3. incorporate communication and caches in the models, lower bounds and the strategies!