# Designing a QR Factorization for Multicore and Multi-GPU Architectures using Runtime Systems

Emmanuel AGULLO (INRIA HiePACS team)
Julien LANGOU (University of Colorado Denver)

Joint work with University of Tennessee and INRIA Runtime

Vers la simulation numérique pétaflopique
sur architectures parallèles hybrides

École CEA-EDF-INRIA, Sophia, France, June 6-10, 2011

INRIA

University of Colorado
Denver | Anschutz Medical Campus

# Motivation

## Algorithm / Software design

"Must rethink the design of our software."
J. Dongarra, yesterday's talk.

## QR factorization

"One algorithmic idea in numerical linear algebra is more important than all the others: QR factorization."
L. N. Trefethen and D. Bau,
Numerical Linear Algebra, SIAM, 1997.

# Motivation

## Algorithm / Software design

"Must rethink the design of our software."
J. Dongarra, yesterday's talk.

## QR factorization

"One algorithmic idea in numerical linear algebra is more important than all the others: QR factorization."
L. N. Trefethen and D. Bau,
Numerical Linear Algebra, SIAM, 1997.

# Three-layers paradigm

## High-level algorithm

## Runtime System

## Device kernels

CPU cores

# Three-layers paradigm

## High-level algorithm

## Runtime System

## Device kernels

CPU core GPU

# Three-layers paradigm

## High-level algorithm

## Runtime System

## Device kernels

CPU core

GPU

# Three-layers paradigm

High-level algorithm

Runtime System

Device kernels

CPU core

GPU

# Three-layers paradigm

## High-level algorithm

## Runtime System

## Device kernels

### CPU core

### GPU

# Objective of the talk

Show that this 3-layers programmation paradigm:

- ⋆ increases productivity;
- ⋆ enables performance portability.

☺ Knowing everything you have ever wondered about QR factorization.

# Objective of the talk

Show that this 3-layers programmation paradigm:

- ⋆ increases productivity;
- ⋆ enables performance portability.

☺ Knowing everything you have ever wondered about QR factorization.

# Outline

1. QR factorization
2. Runtime System for multicore architectures
3. Using Accelerators
4. Enhancing parallelism
5. Distributed Memory

# Outline

# Three-layers paradigm

## High-level algorithm

QR factorization

## Runtime System

## Device kernels

CPU core

GPU

# Motivation for tile algorithms

Need to design new algorithm in order to

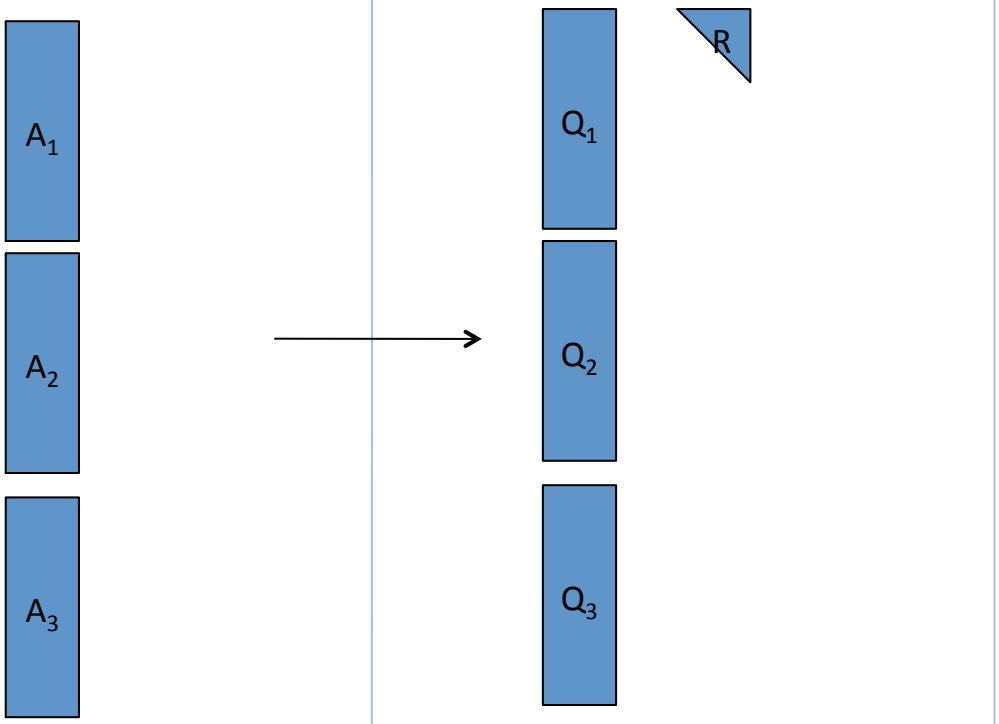
- reduce communication
- enhance parallelism

# 1D tile QR - binary tree

## Reduce Algorithms: Introduction

The QR factorization of a long and skinny matrix with its data partitioned vertically across several processors arises in a wide range of applications.

| **Input**: | **Output:** |
| --- | --- |
| A is block distributed by rows | Q is block distributed by rows<br>R is global |



# 1D tile QR - binary tree

## Reduce Algorithms: Introduction

**Example of applications:**

a) in linear least squares problems which the number of equations is extremely larger than the number of unknowns

b) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers)

c) in dense large and more square QR factorization where they are used as the panel factorization step

# 1D tile QR - binary tree

## Reduce Algorithms: Introduction

**Example of applications:**

a) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers),

b) in dense large and more square QR factorization where they are used as the panel factorization step, or more simply

c) in linear least squares problems which the number of equations is extremely larger than the number of unknowns.

**The main characteristics of those three examples are that**

a) there is **only one column of processors involved** but several processor rows,

b) **all the data is known from the beginning**,

c) and **the matrix is dense**.

**Various methods already exist** to perform the QR factorization of such matrices:

a) Gram-Schmidt (**mgs(row)**,**cgs**),

b) Householder (**qr2**, **qrf**),

c) or **CholeskyQR**.

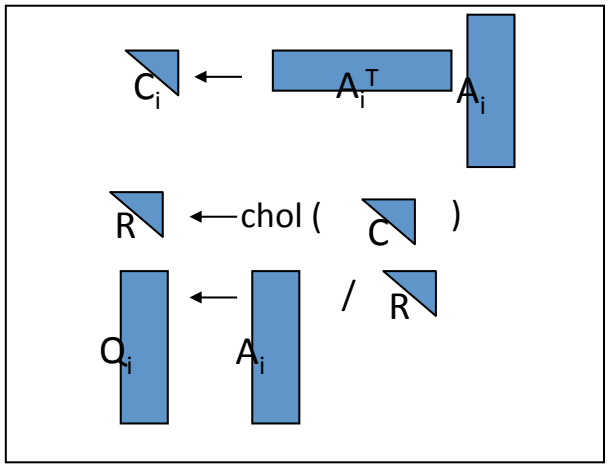
We present a new method:

Allreduce Householder (**rhh_qr3**, **rhh_qrf**).

# 1D tile QR - binary tree

## The CholeskyQR Algorithm

| | | |
|---|---|---|
| **SYRK:** | $C := A^T A$ | $( mn^2 )$ |
| **CHOL:** | $R := chol( C )$ | $( n^3/3 )$ |
| **TRSM:** | $Q := A/R$ | $( mn^2 )$ |



# 1D tile QR - binary tree

## Bibligraphy


- A. Stathopoulos and K. Wu, A block orthogonalization procedure with constant synchronization requirements, *SIAM Journal on Scientific Computing*, 23(6):2165-2182, 2002.

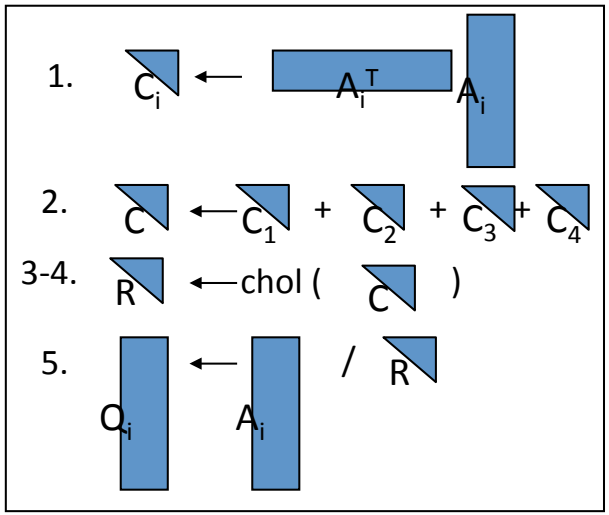
- Popularized by iterative eigensolver libraries:
  1) PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
  2) HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
  3) Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
  4) PRIMME (A. Stathopoulos, Coll. William & Mary ).

# 1D tile QR - binary tree

## Parallel distributed CholeskyQR

The CholeskyQR method in the parallel distributed context can be described as follows:

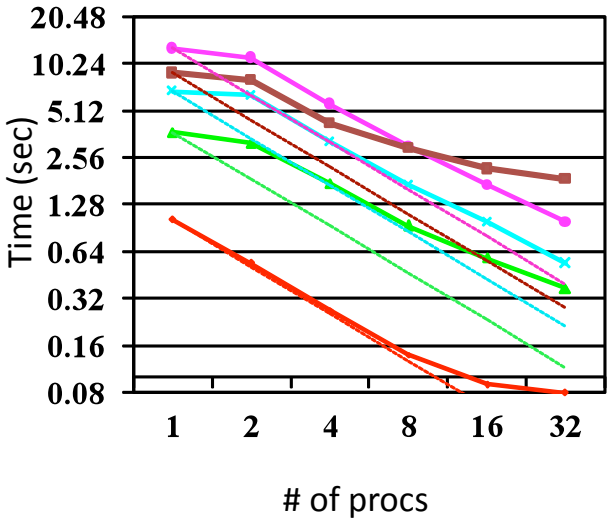
| | | |
|---|---|---|
| 1: SYRK: | $C := A^T A$ | $( mn^2 )$ |
| 2: MPI_Reduce: | $C := sum_{procs}\ C$ | (on proc 0) |
| 3: CHOL: | $R := chol( C )$ | $( n^3/3 )$ |
| 4: MPI_Bdcast | Broadcast the R factor on proc 0 | |
| | to all the other processors | |
| 5: TRSM: | $Q := A/R$ | $( mn^2 )$ |

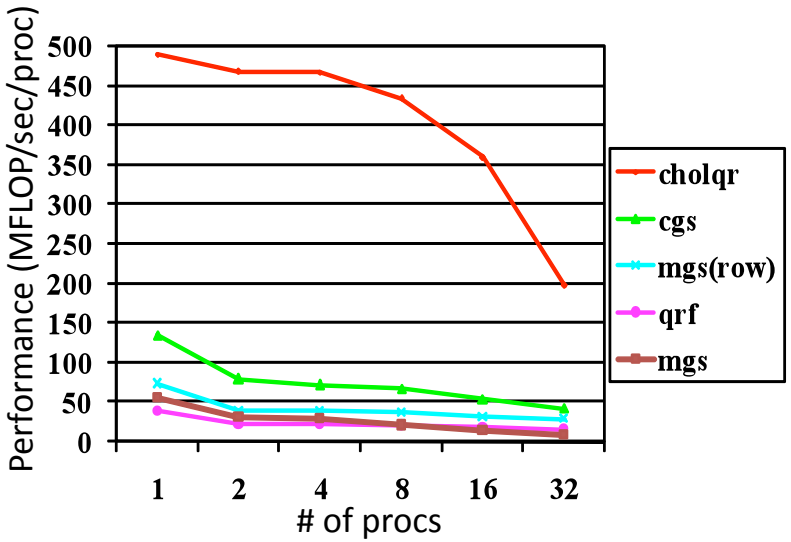


# 1D tile QR - binary tree

In this experiment, we fix the problem: **m=100,000** and **n=50**.

## Efficient enough?



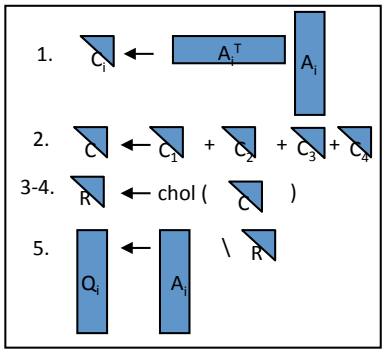

| # of procs | cholqr | | cgs | | mgs(row) | | qrf | | mgs | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 489.2 | (1.02) | 134.1 | (3.73) | 73.5 | (6.81) | 39.1 | (12.78) | 56.18 | (8.90) |
| 2 | 467.3 | (0.54) | 78.9 | (3.17) | 39.0 | (6.41) | 22.3 | (11.21) | 31.21 | (8.01) |
| 4 | 466.4 | (0.27) | 71.3 | (1.75) | 38.7 | (3.23) | 22.2 | (5.63) | 29.58 | (4.23) |
| 8 | 434.0 | (0.14) | 67.4 | (0.93) | 36.7 | (1.70) | 20.8 | (3.01) | 21.15 | (2.96) |
| 16 | 359.2 | (0.09) | 54.2 | (0.58) | 31.6 | (0.99) | 18.3 | (1.71) | 14.44 | (2.16) |
| 32 | 197.8 | (0.08) | 41.9 | (0.37) | 29.0 | (0.54) | 15.8 | (0.99) | 8.38 | (1.87) |

MFLOP/sec/proc

Time in sec

# 1D tile QR - binary tree

## Simple enough?



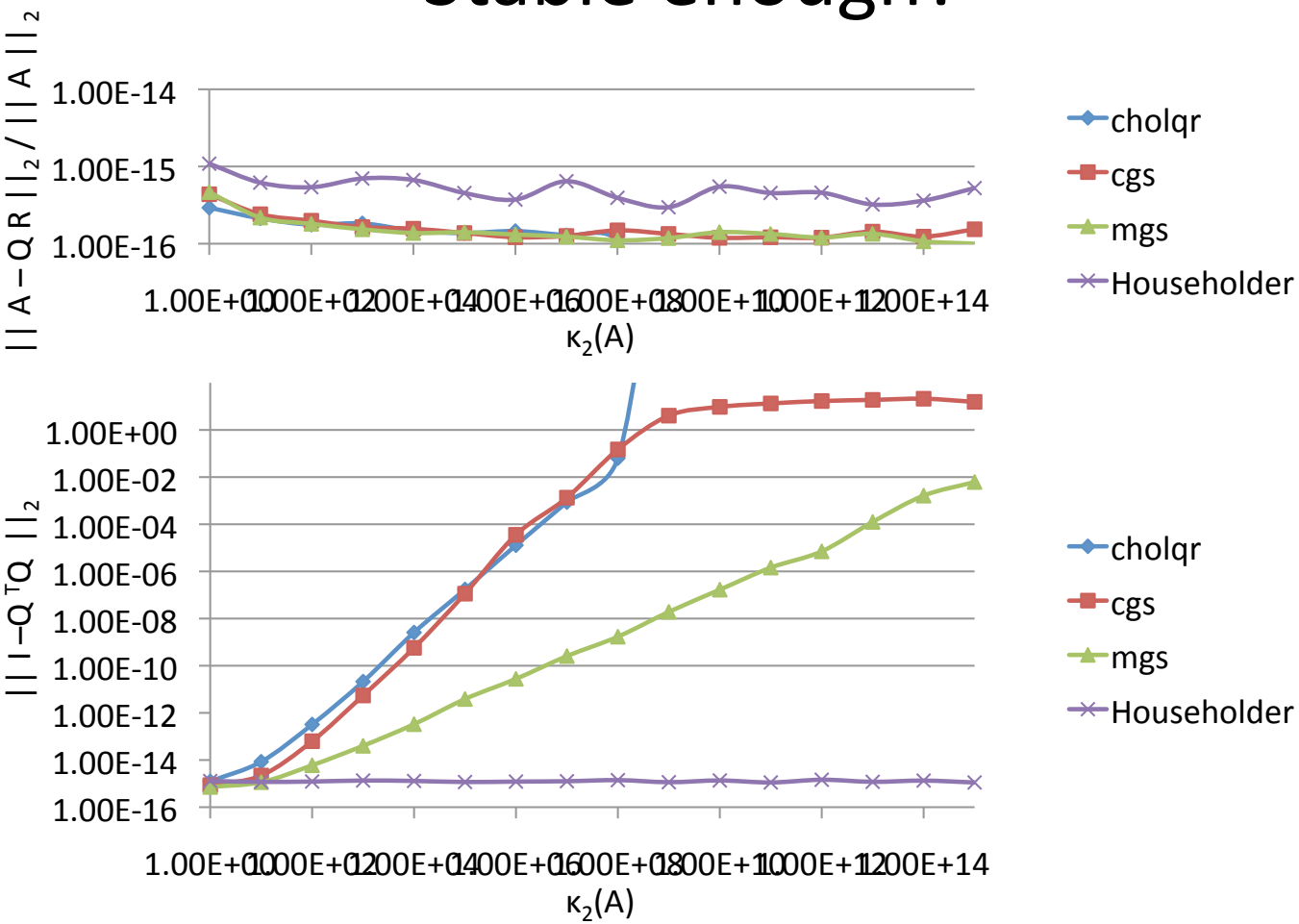

```
int choleskyqr_A_v0(int mloc, int n, double *A, int lda, double *R, int ldr,
                    MPI_Comm mpi_comm){

                    int info;
                    cblas_dsyrk( CblasColMajor, CblasUpper, CblasTrans, n, mloc,
                                 1.0e+00, A, lda, 0e+00, R, ldr );
                    MPI_Allreduce( MPI_IN_PLACE, R, n*n, MPI_DOUBLE, MPI_SUM, mpi_comm );
                    lapack_dpotrf( lapack_upper, n, R, ldr, &info );
                    cblas_dtrsm( CblasColMajor, CblasRight, CblasUpper, CblasNoTrans, CblasNonUnit,
                                 mloc, n, 1.0e+00, R, ldr, A, lda );
                    return 0;

}
```

( … and, OK, you might want to add an MPI user defined datatype to send only the upper part of R)

# 1D tile QR - binary tree

m=100, n=50

## Stable enough?

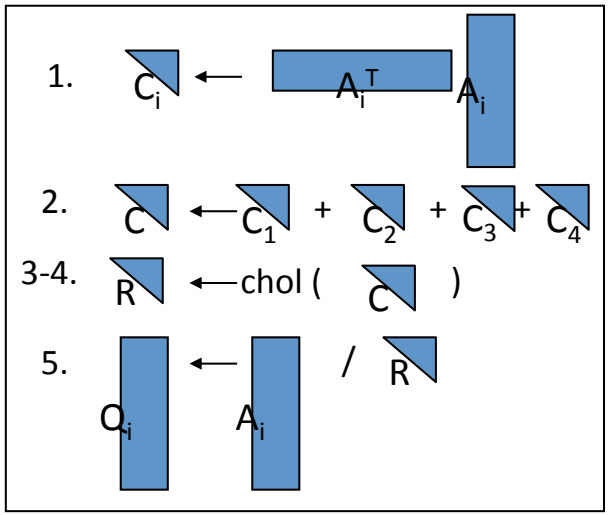


# 1D tile QR - binary tree

## Parallel distributed CholeskyQR

The CholeskyQR method in the parallel distributed context can be described as follows:

| | | |
|---|---|---|
| 1: SYRK: | C:= $A^TA$ | ( $mn^2$) |
| 2: MPI_Reduce: | C:= $sum_{procs}$ C | (on proc 0) |
| 3: CHOL: | R := chol( C ) | ( $n^3/3$ ) |
| 4: MPI_Bdcast | Broadcast the R factor on proc 0 | |
| | to all the other processors | |
| 5: TRSM: | Q := A/R | ( $mn^2$) |




This method is extremely fast. For two reasons:

1. first, there is **only one or two communications phase**,
2. second, **the local computations are performed with fast operations**.

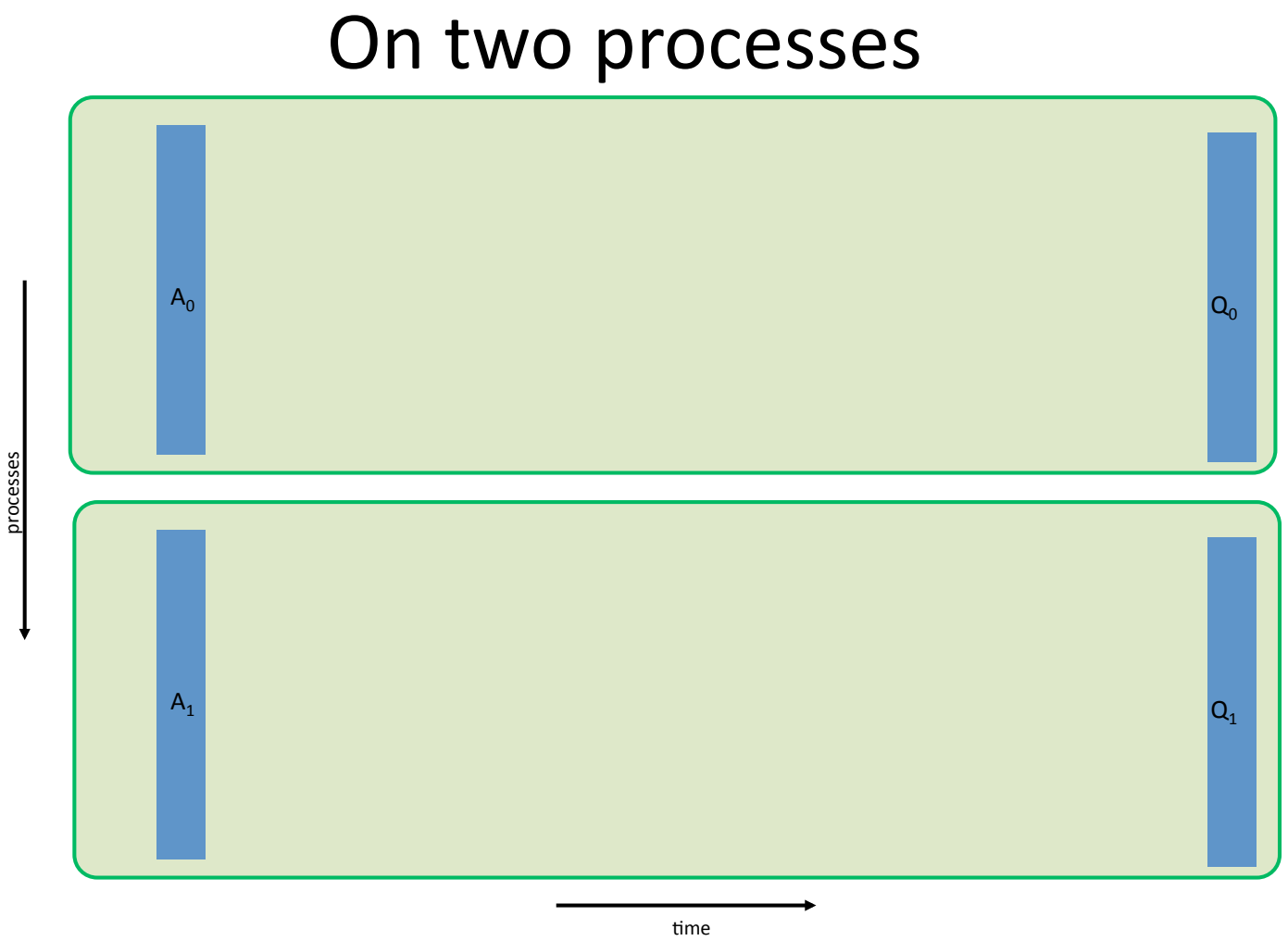
Another advantage of this method is that the resulting code is exactly four lines,

3. so the method is **simple** and relies heavily on other libraries.

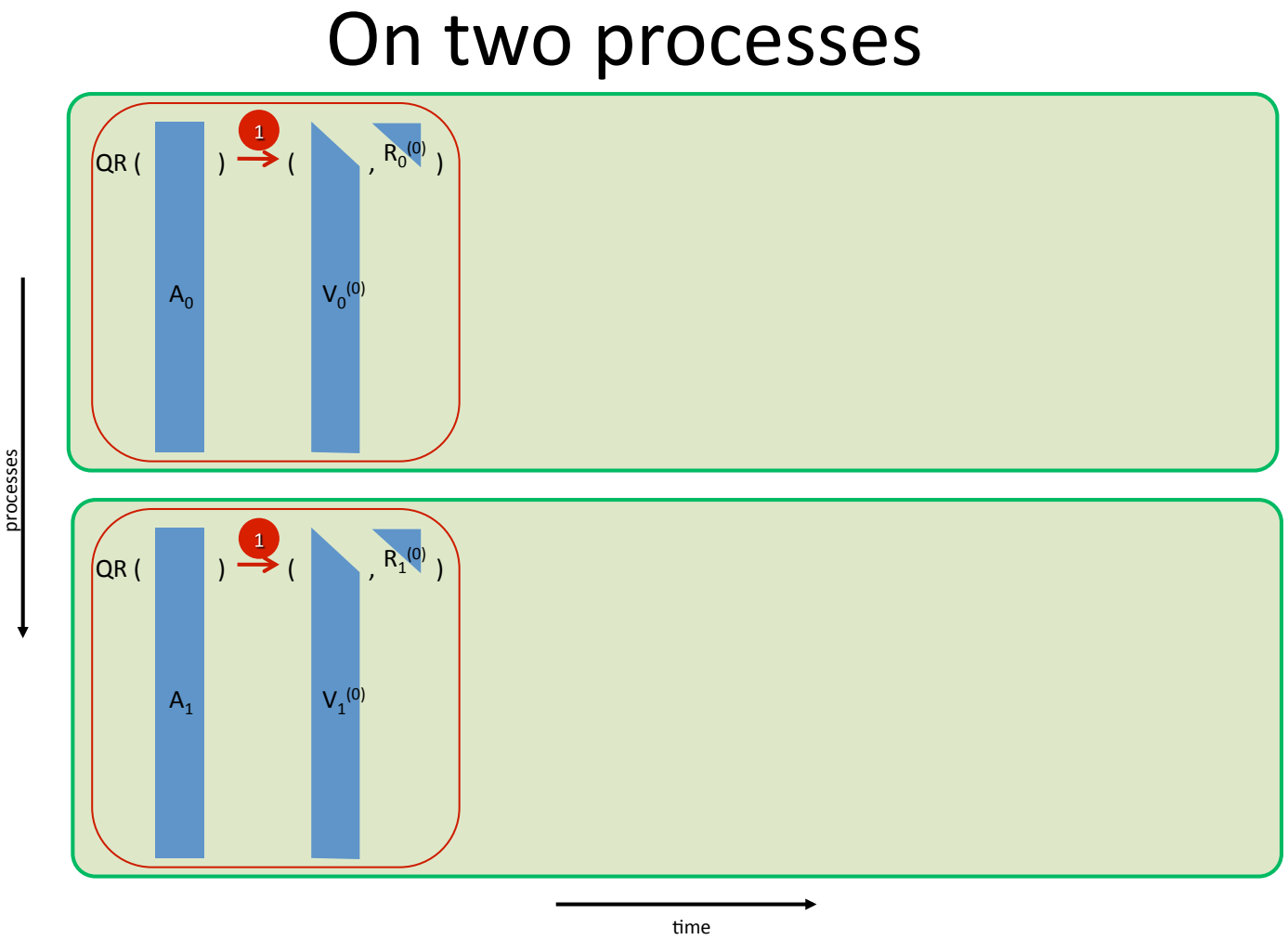
Despite all those advantages,

4. this method is **highly unstable**.

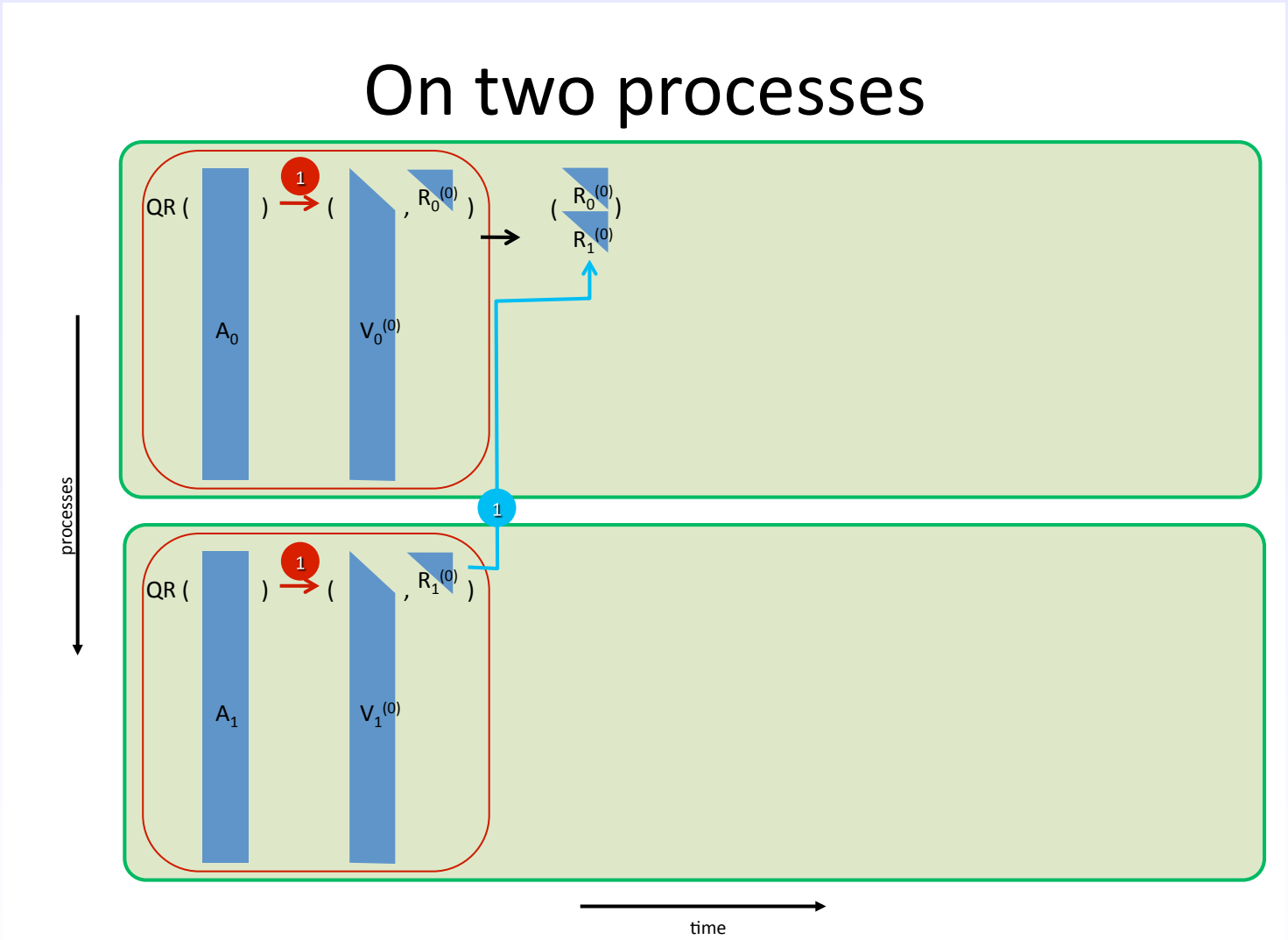
# 1D tile QR - binary tree



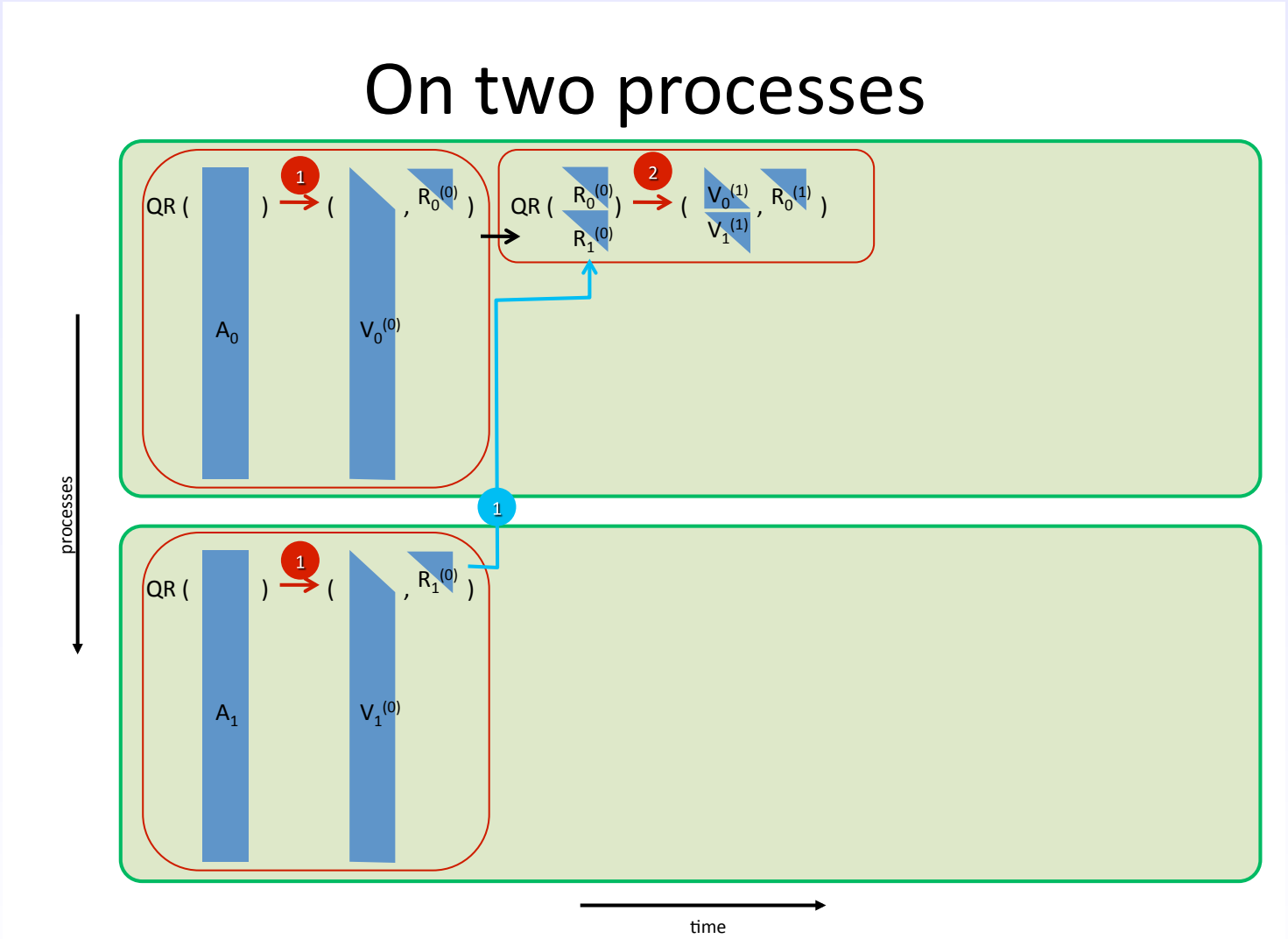
# 1D tile QR - binary tree



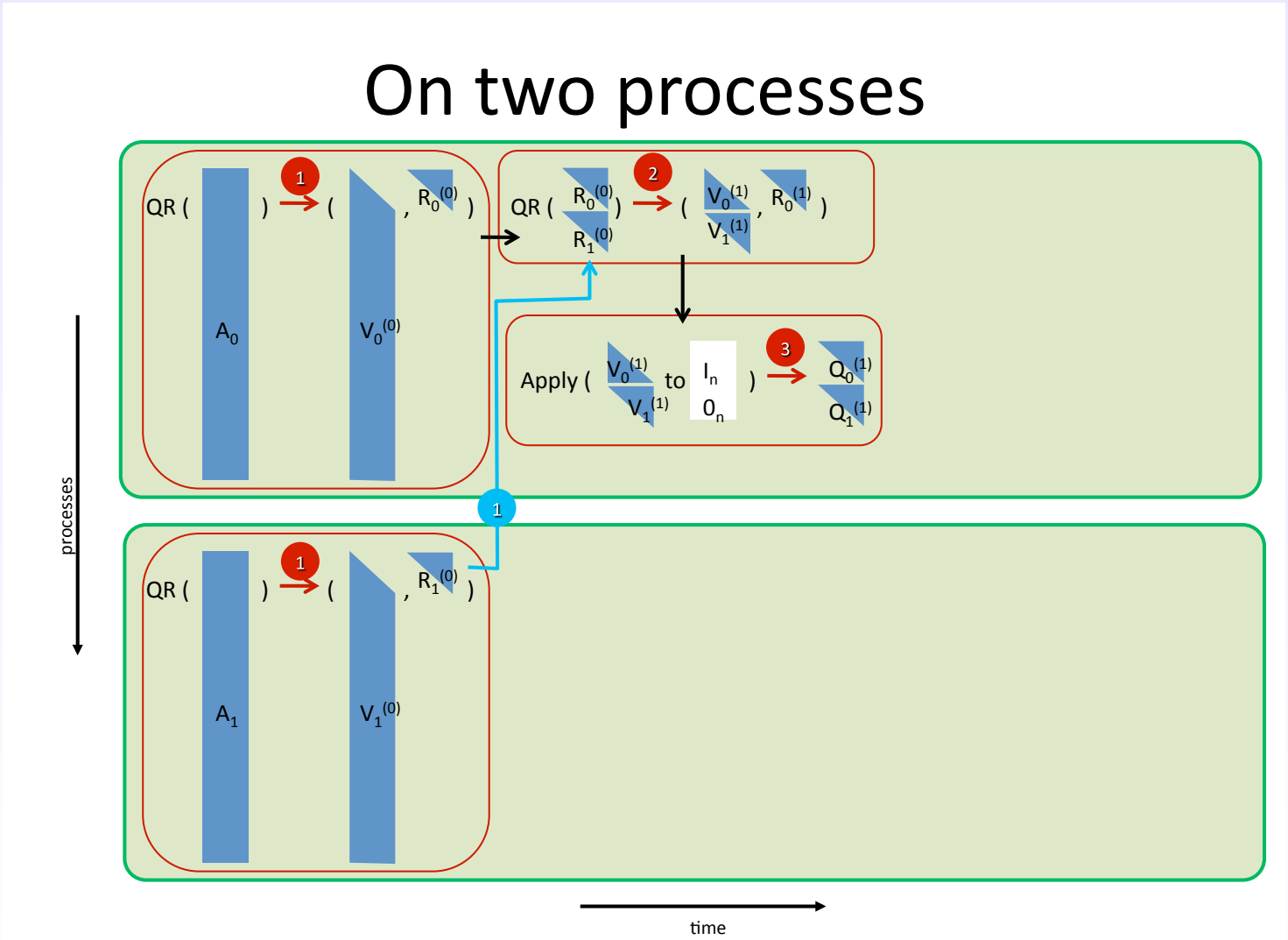
# 1D tile QR - binary tree



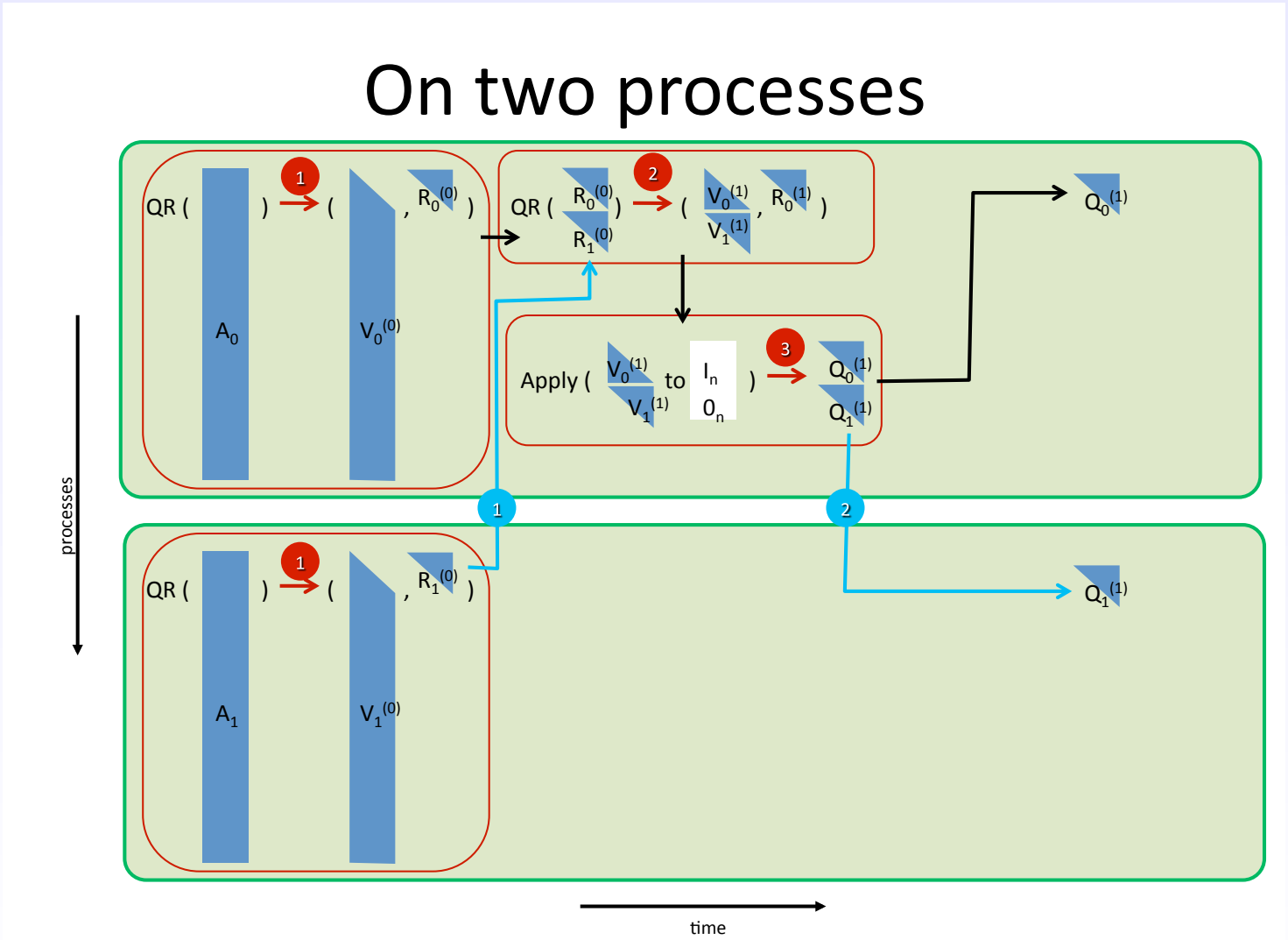
# 1D tile QR - binary tree



# 1D tile QR - binary tree

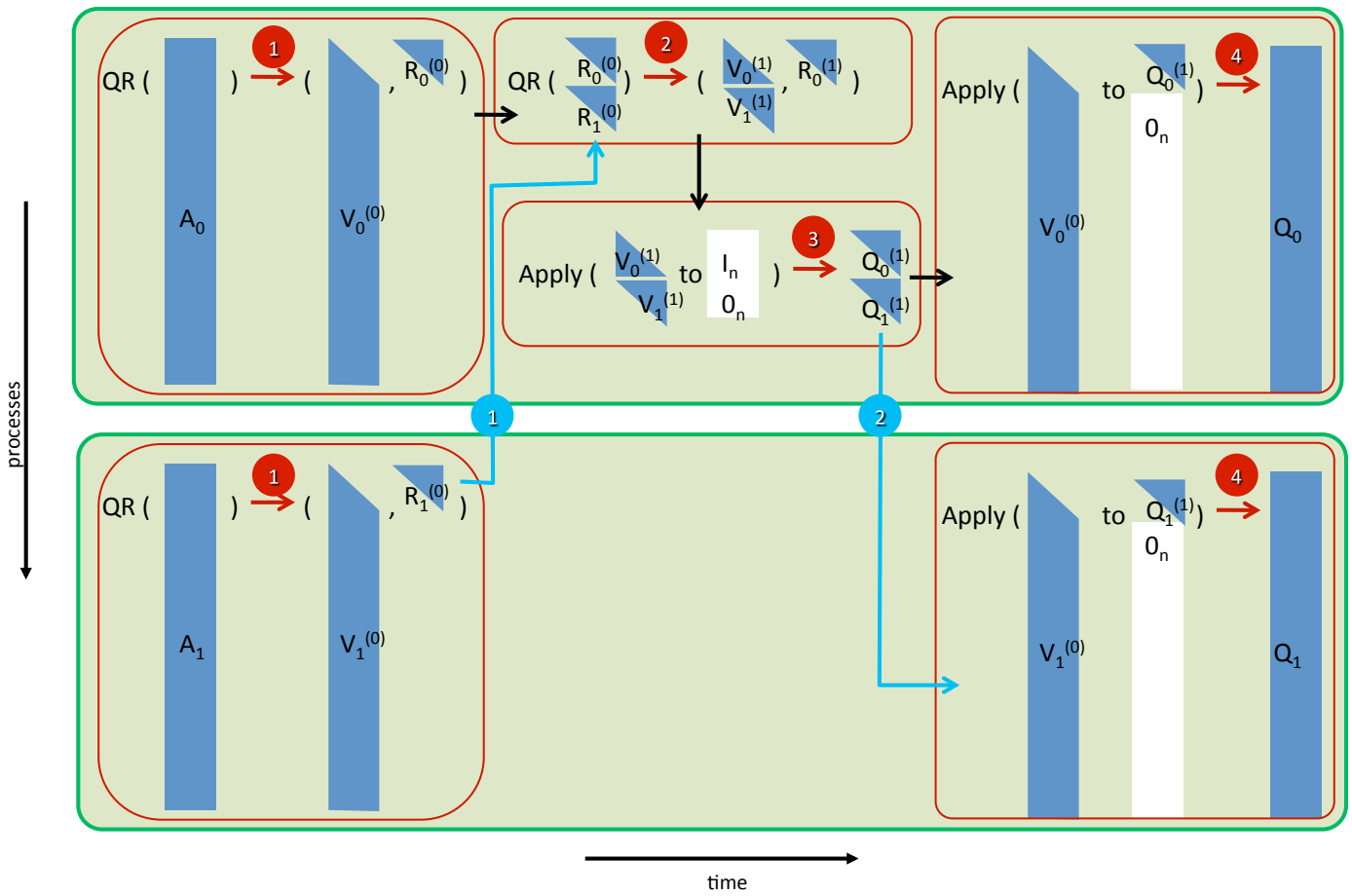


# 1D tile QR - binary tree

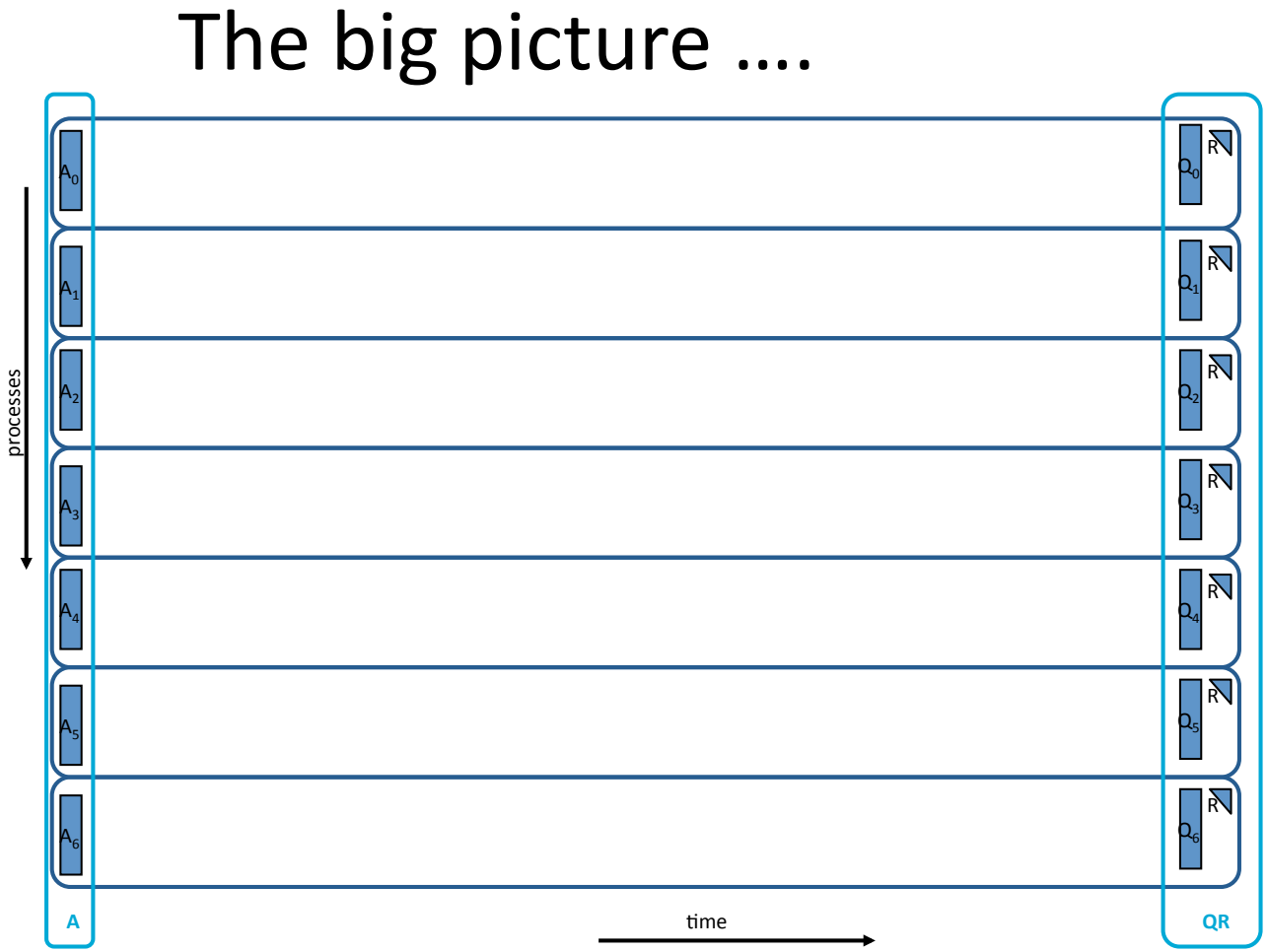


# 1D tile QR - binary tree

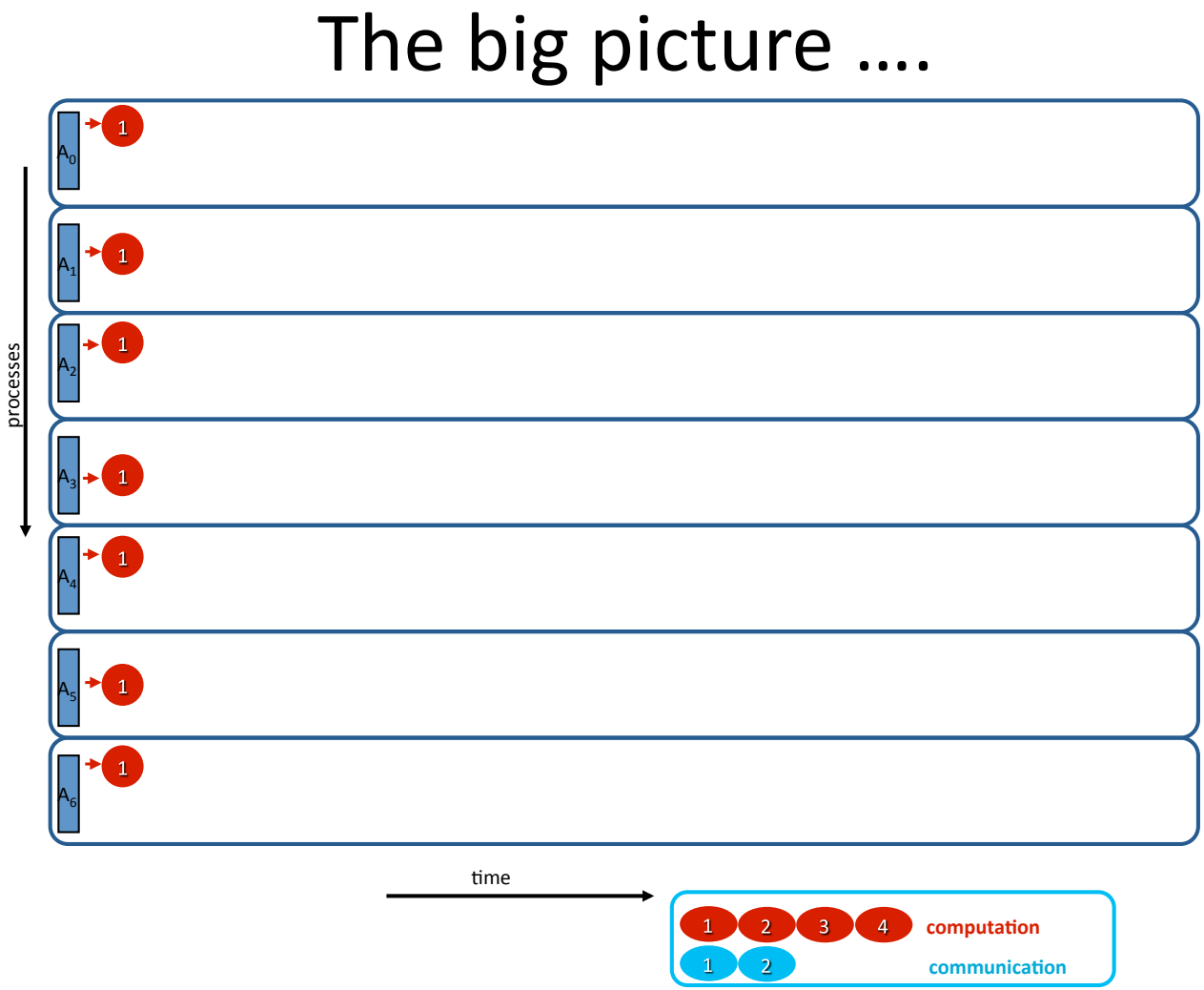
## On two processes



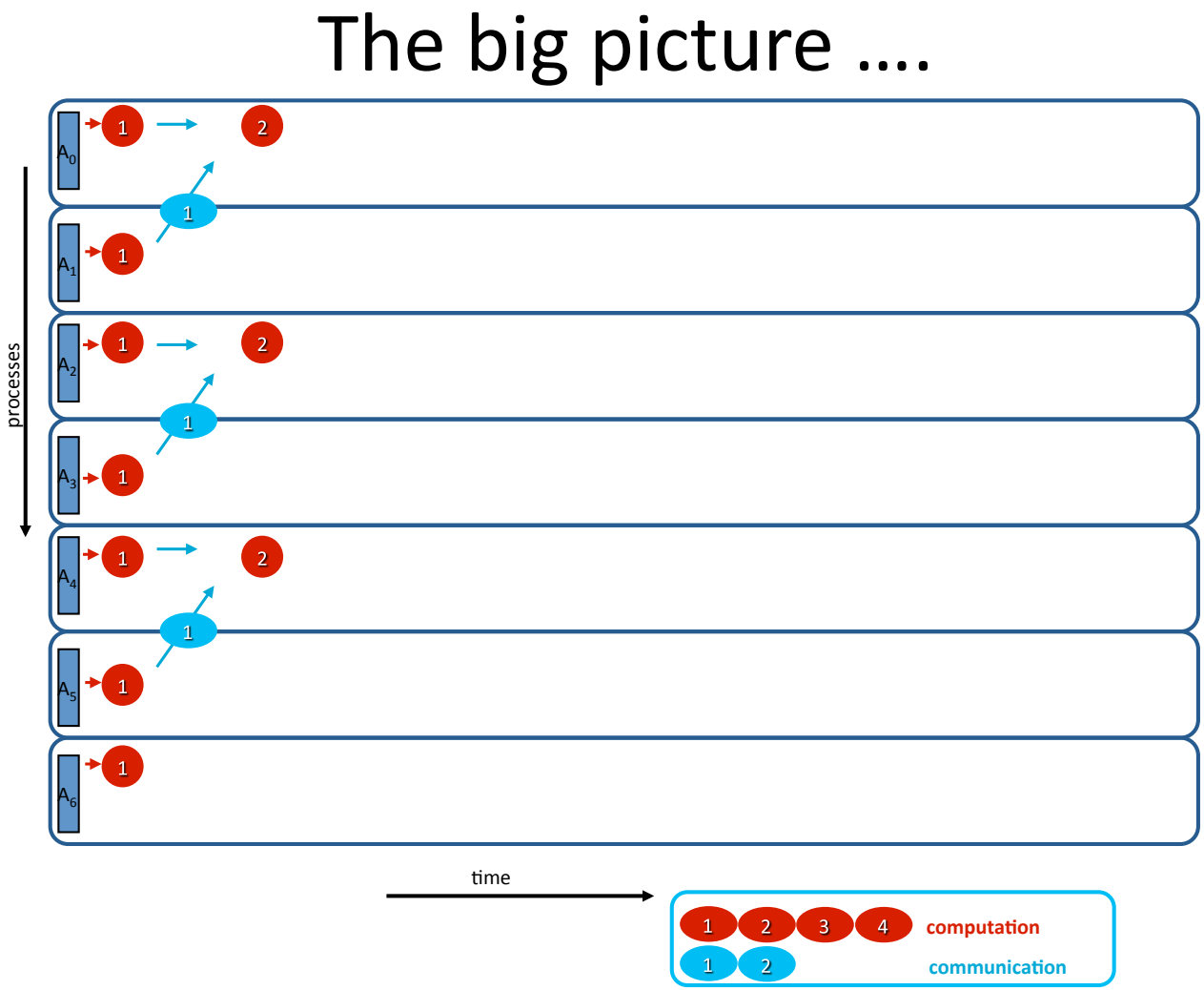
# 1D tile QR - binary tree



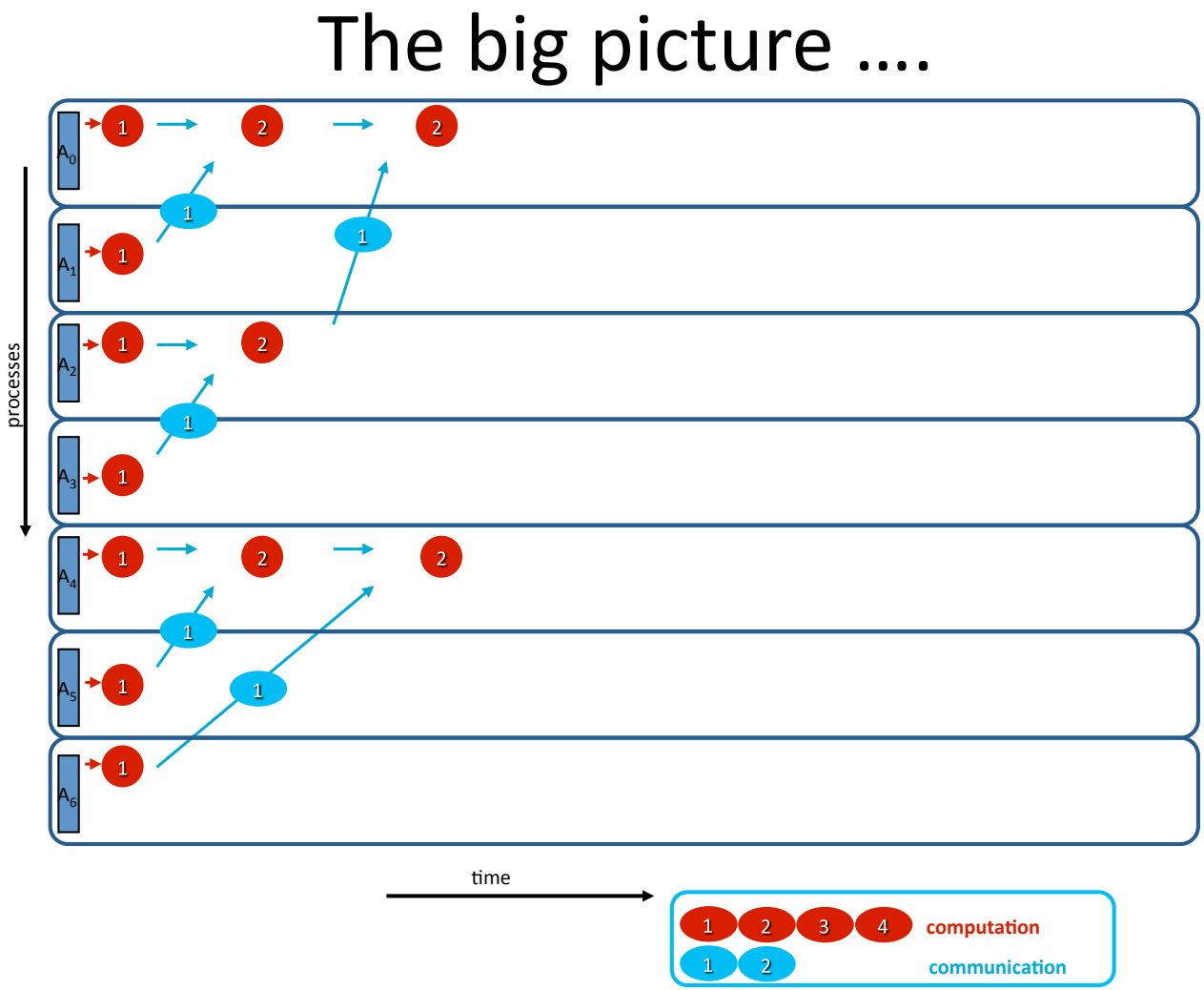
# 1D tile QR - binary tree



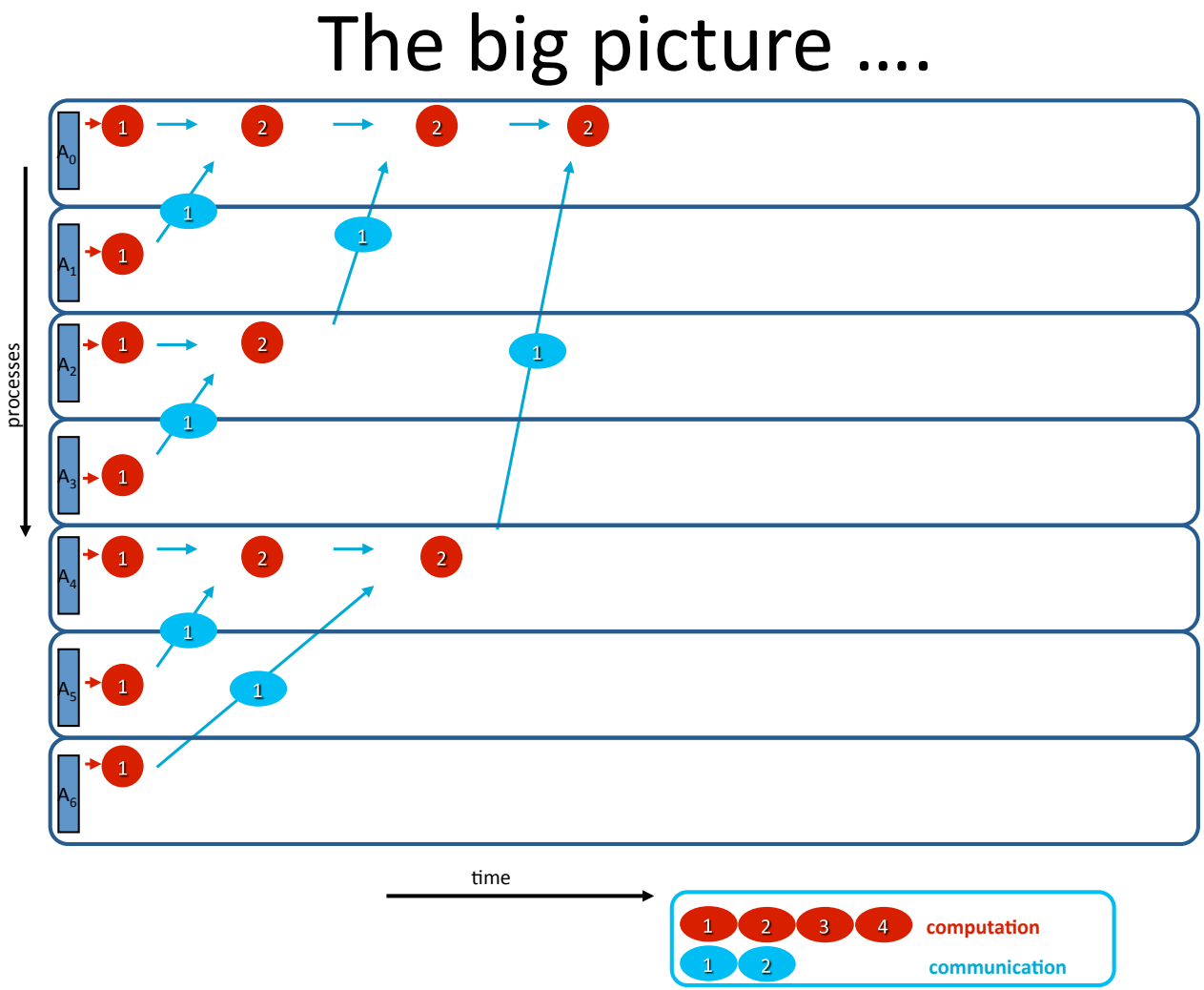
# 1D tile QR - binary tree



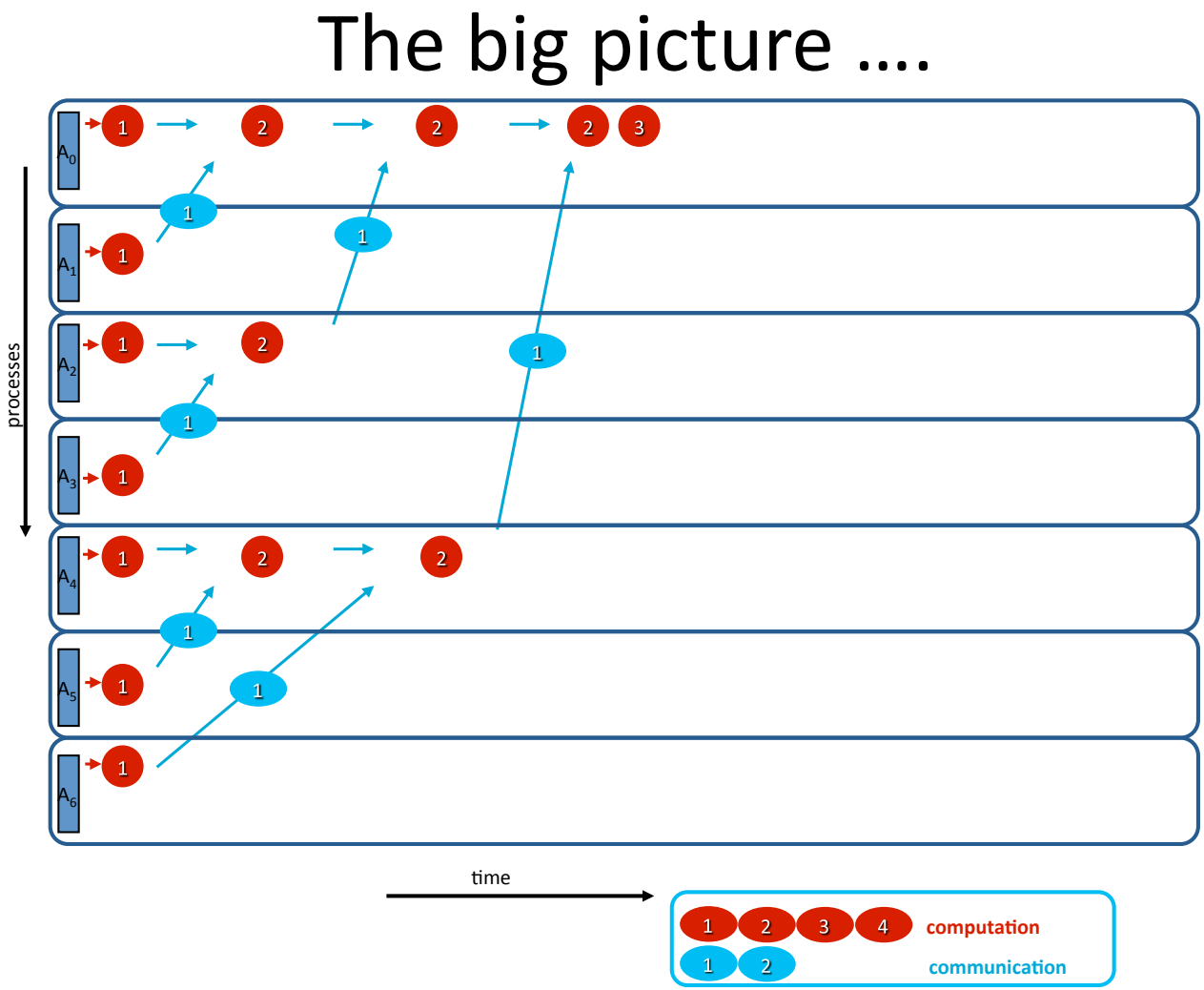
# 1D tile QR - binary tree



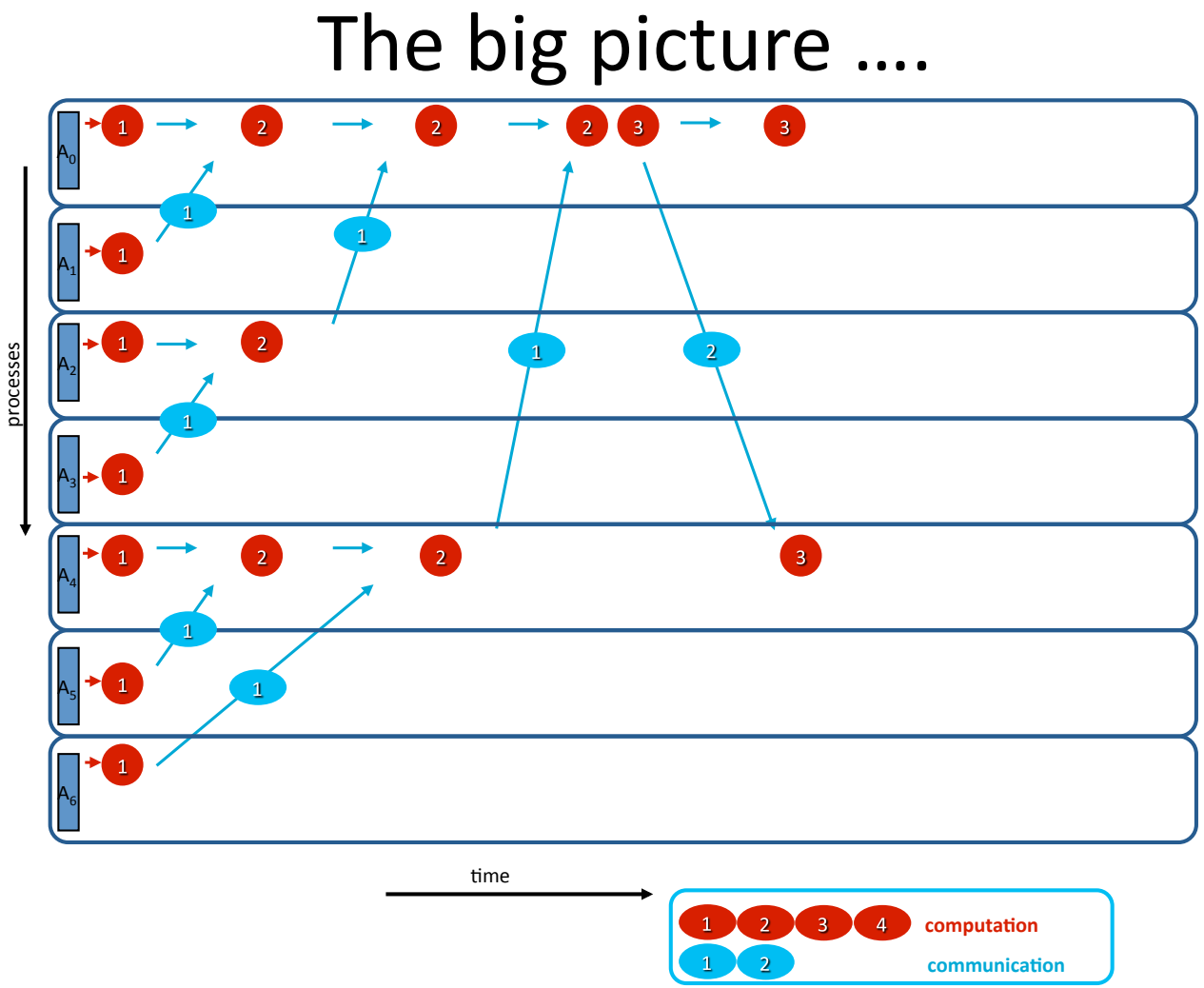
# 1D tile QR - binary tree



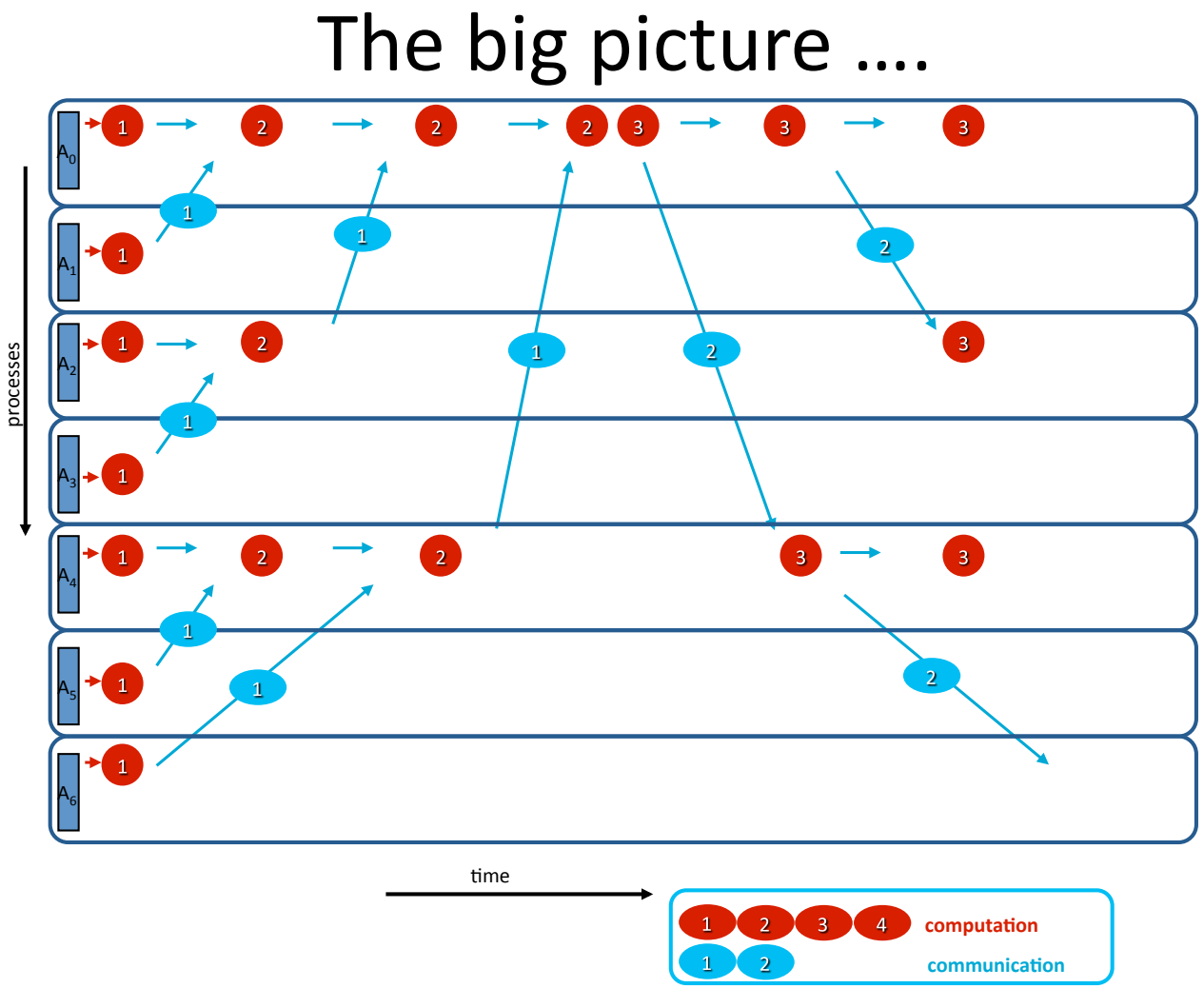
# 1D tile QR - binary tree



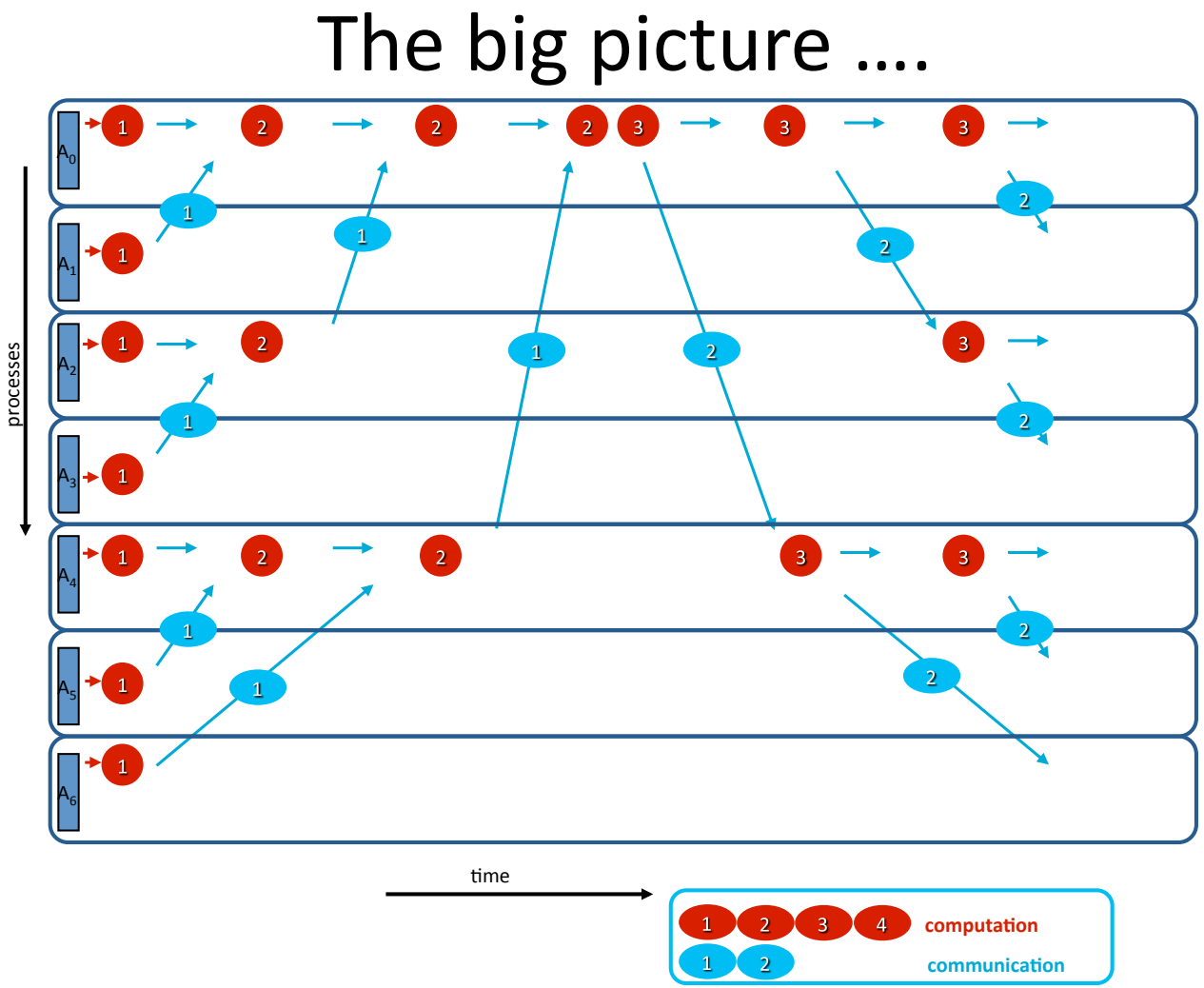
# 1D tile QR - binary tree



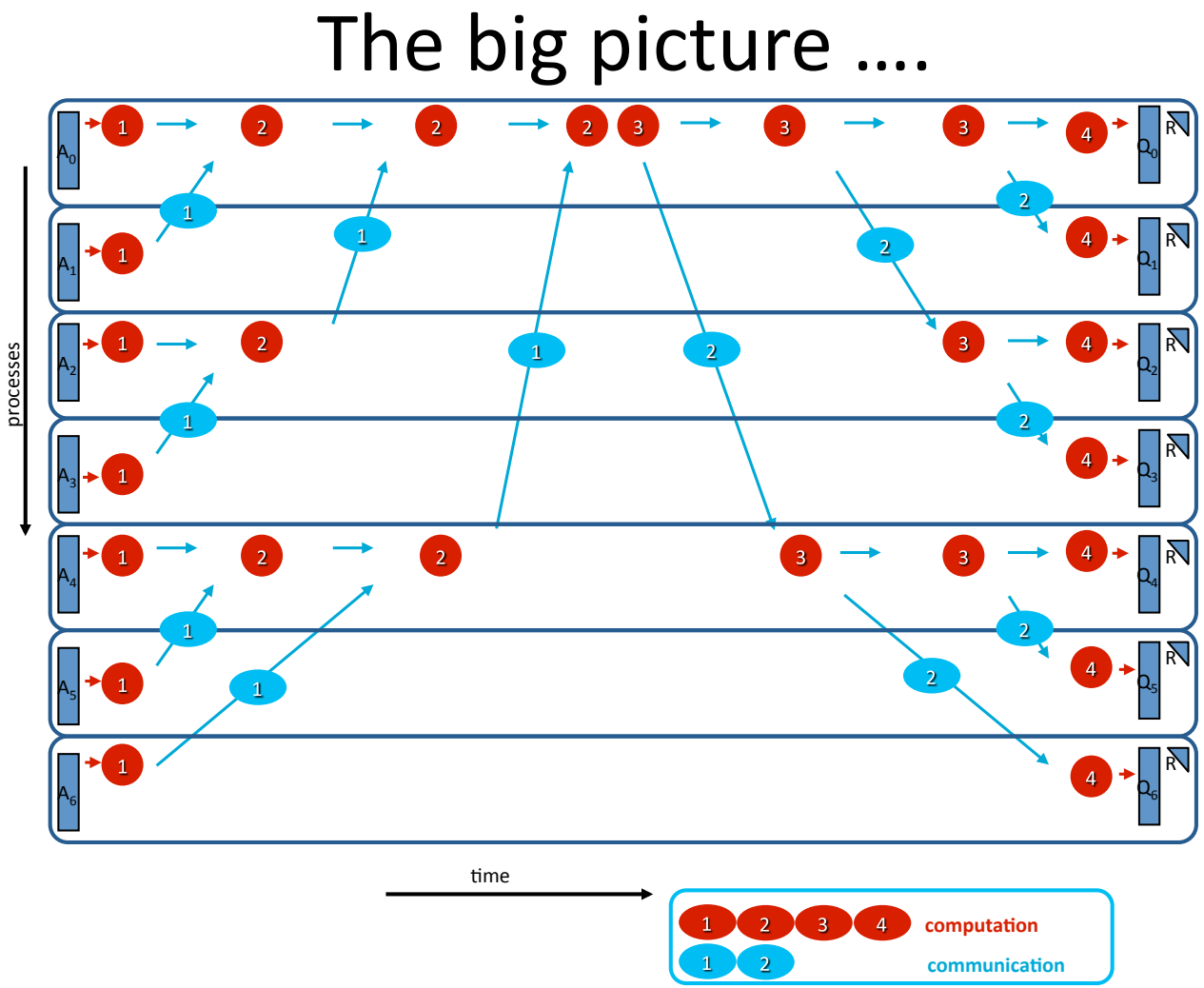
# 1D tile QR - binary tree



# 1D tile QR - binary tree



# 1D tile QR - binary tree

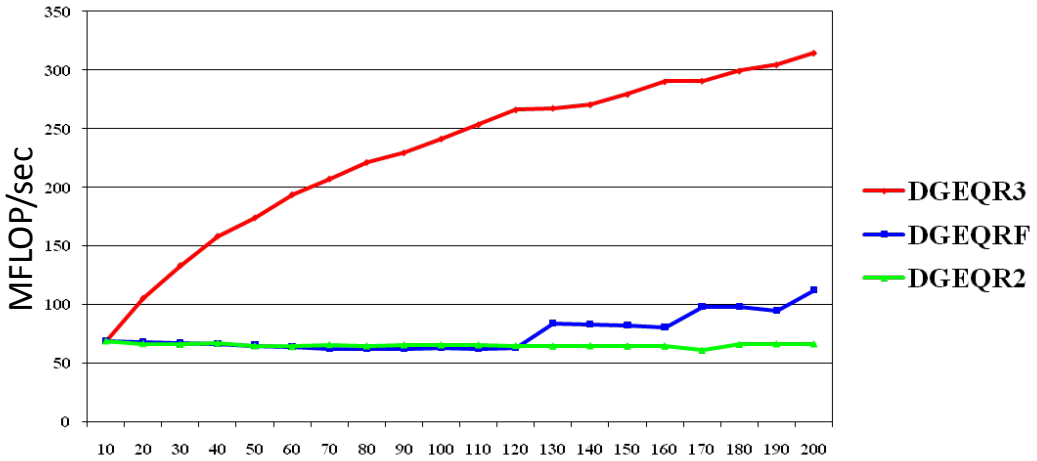


# 1D tile QR - binary tree

## Latency but also possibility of fast panel factorization.

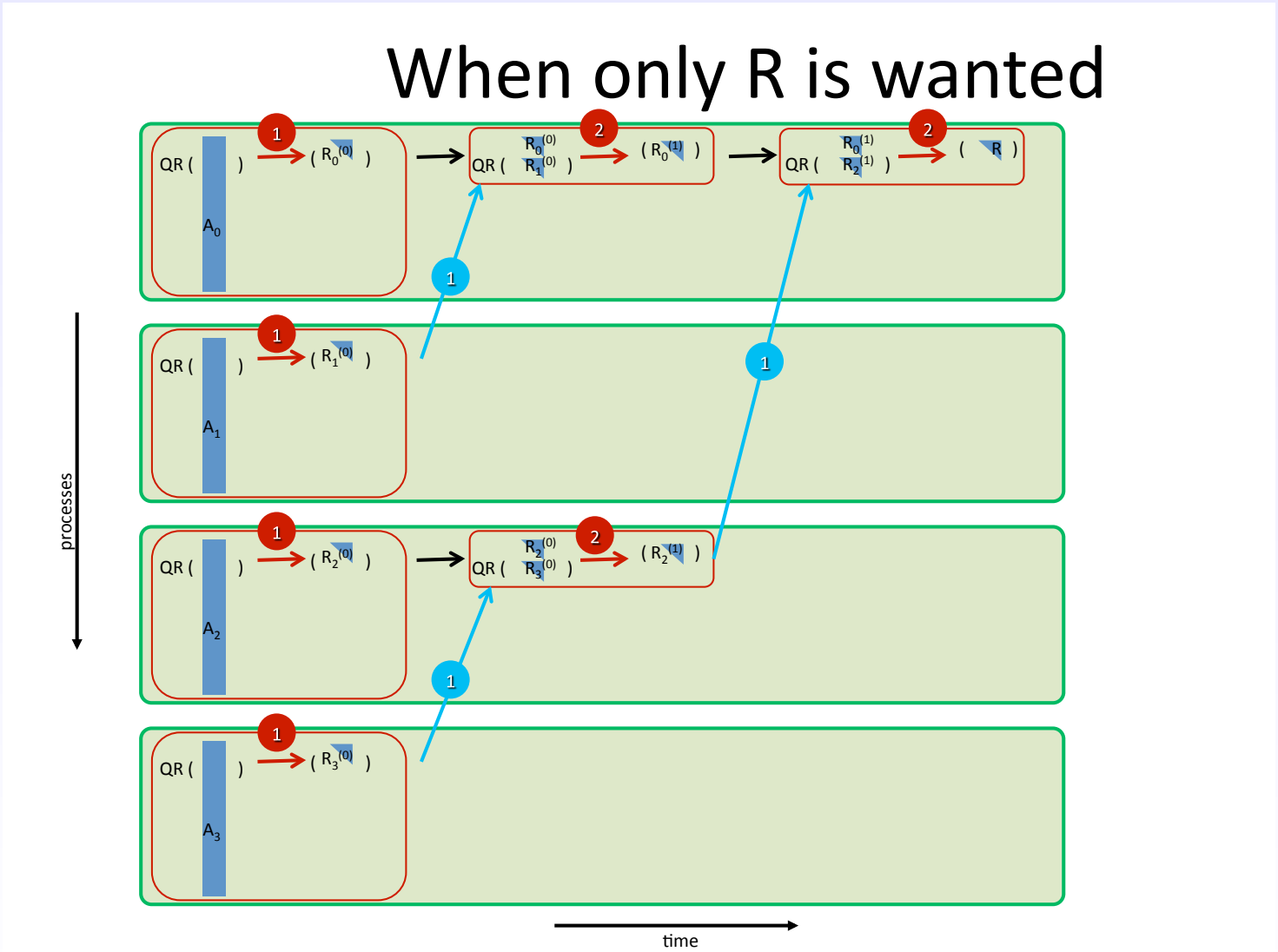
- **DGEQR3** is the recursive algorithm (see Elmroth and Gustavson, 2000), **DGEQRF** and **DGEQR2** are the LAPACK routines.
- Times include QR and DLARFT.
- Run on Pentium III.

| QR factorization and construction of T<br>m = 10,000<br>Perf in MFLOP/sec (Times in sec) | | | | | | |
|---|---|---|---|---|---|---|
| n | DGEQR3 | | DGEQRF | | DGEQR2 | |
| 50 | 173.6 | (0.29) | 65.0 | (0.77) | 64.6 | (0.77) |
| 100 | 240.5 | (0.83) | 62.6 | (3.17) | 65.3 | (3.04) |
| 150 | 277.9 | (1.60) | 81.6 | (5.46) | 64.2 | (6.94) |
| 200 | 312.5 | (2.53) | 111.3 | (7.09) | 65.9 | (11.98) |




m=1000,000, the x axis is n

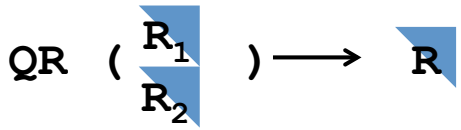
# 1D tile QR - binary tree



# 1D tile QR - binary tree

## 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:




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:

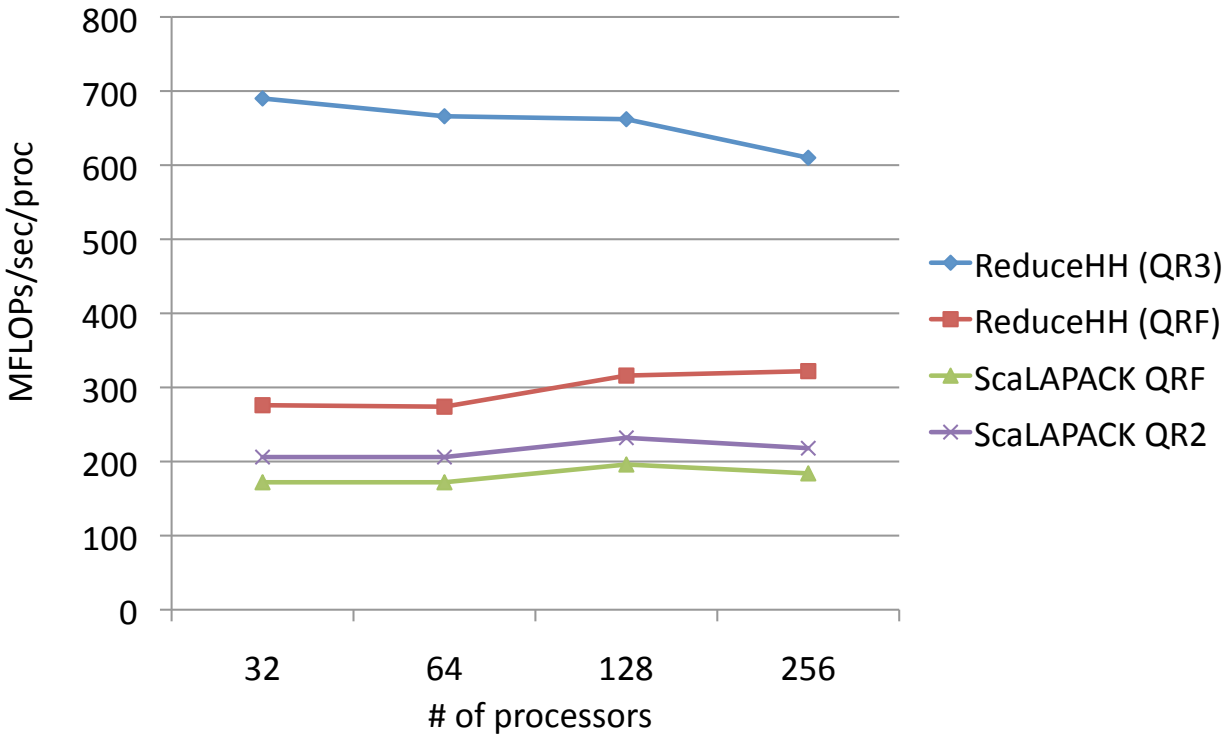
```
lapack_dgeqrf( mloc, n, A, lda, tau, &dlwork, lwork, &info );
MPI_Allreduce( MPI_IN_PLACE, A, 1, MPI_UPPER,
        LILA_MPIOP_QR_UPPER, mpi_comm);
```

# 1D tile QR - binary tree

Blue Gene L
frost.ncar.edu

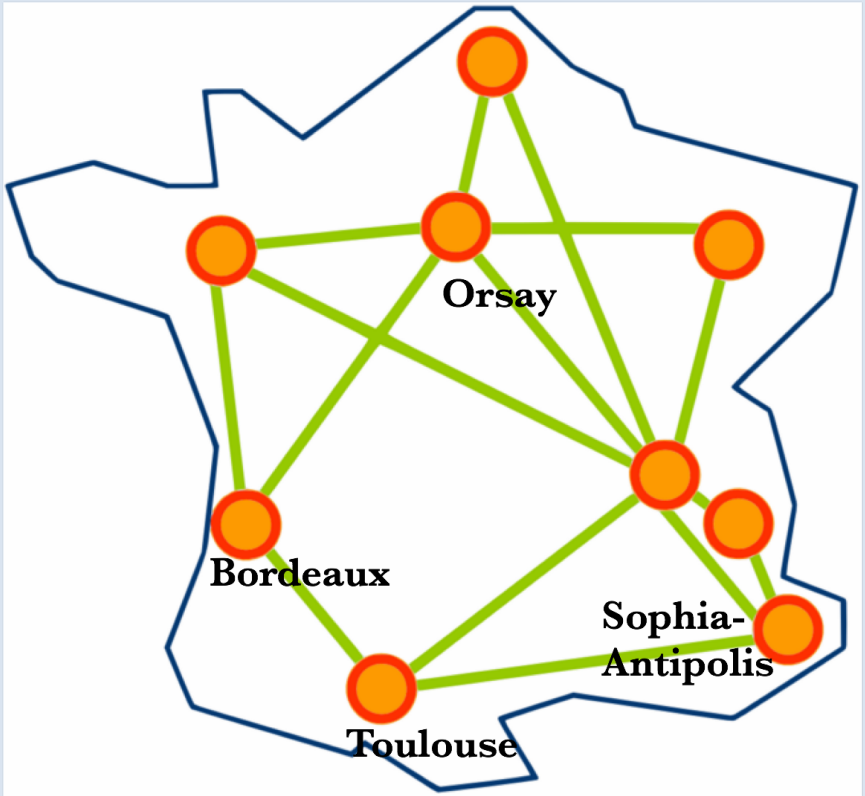
## Q and R: Strong scalability

In this experiment, we fix the problem: **m=1,000,000** and **n=50**.
Then we increase the number of processors.



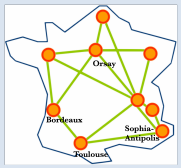
# 1D tile QR - binary tree

## Grid 5000



# 1D tile QR - binary tree

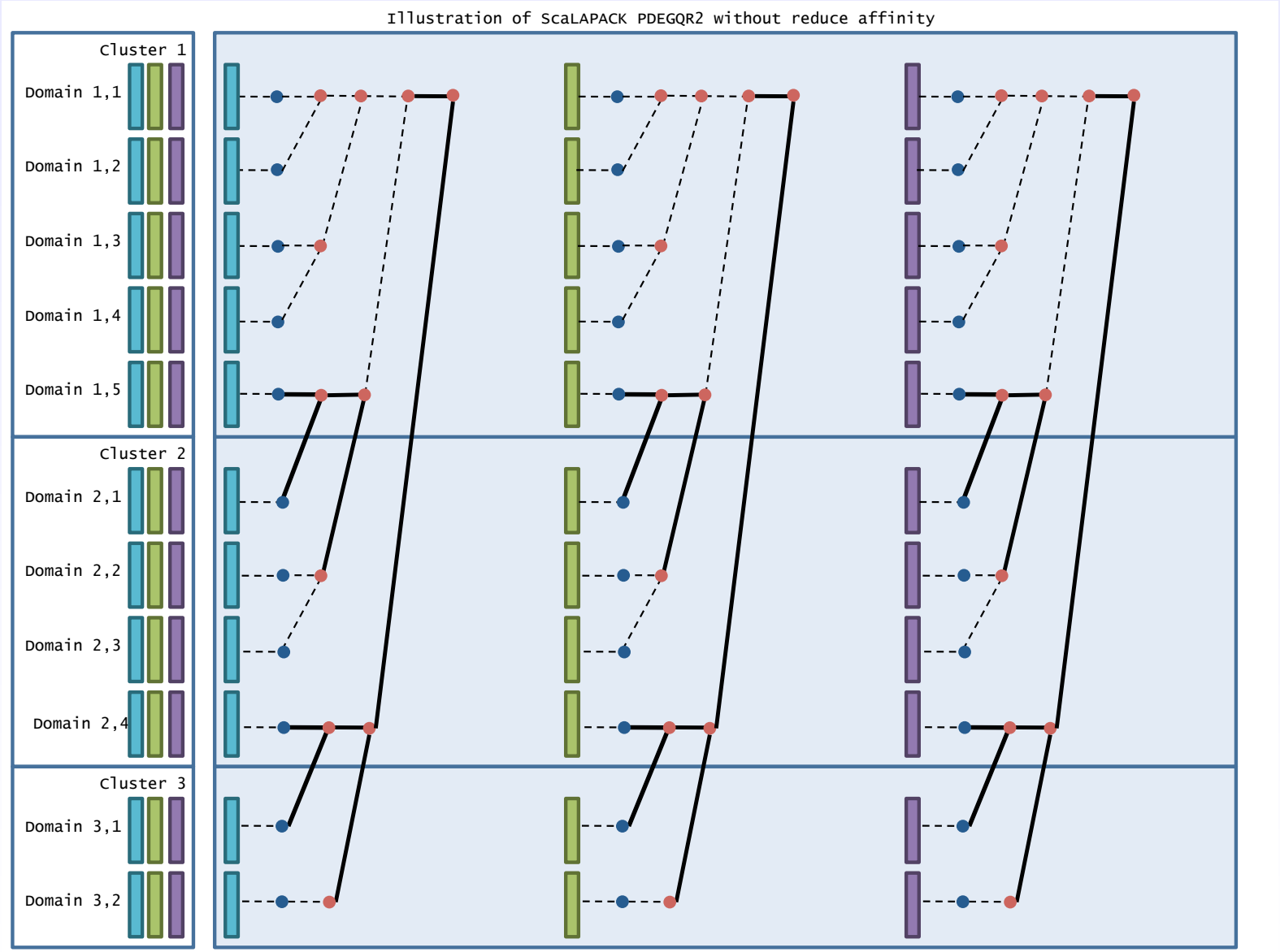
## Grid 5000



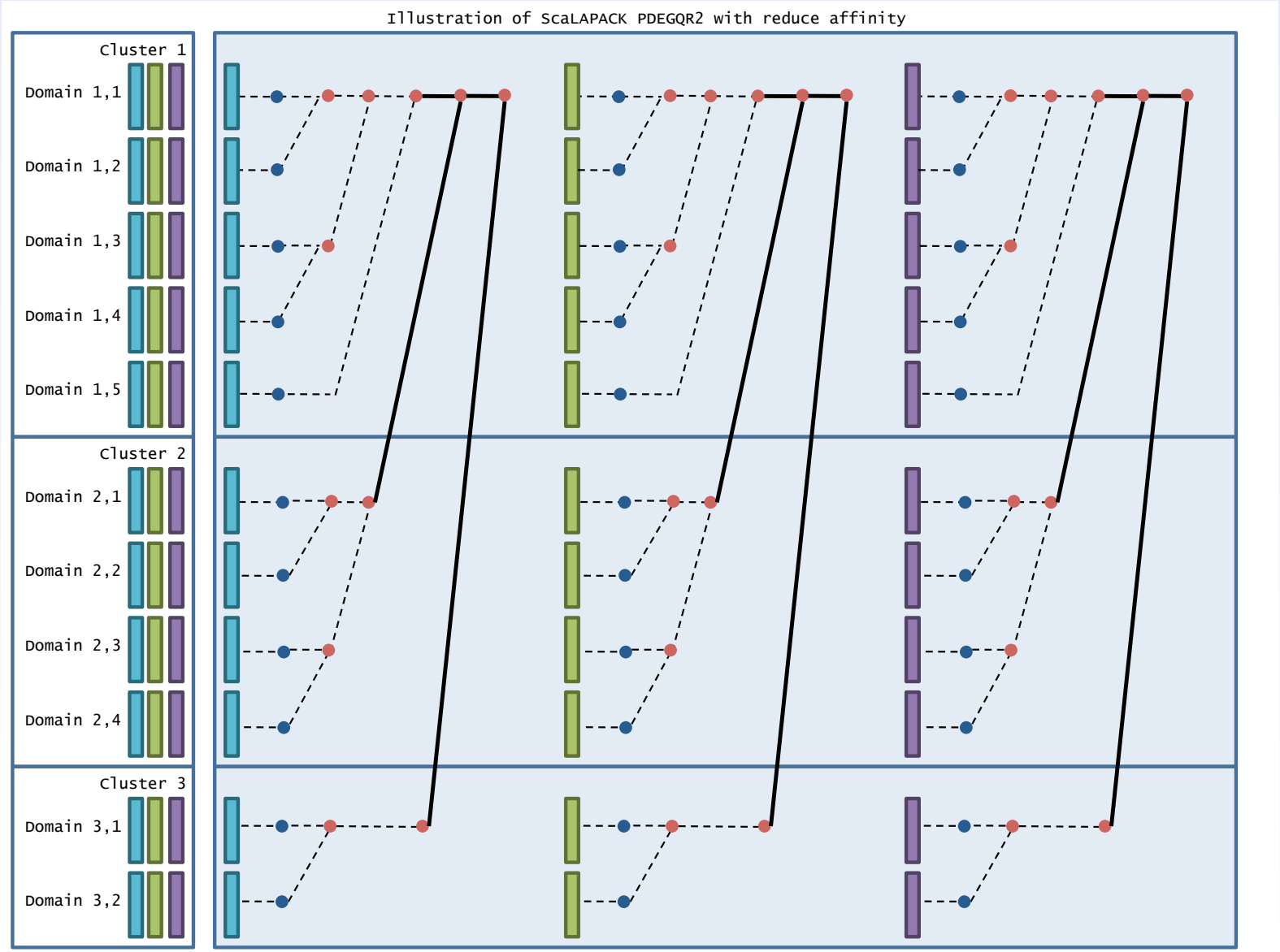

| Latency (ms) | Orsay | Toulouse | Bordeaux | Sophia |
|---|---|---|---|---|
| Orsay | 0.07 | 7.97 | 6.98 | 6.12 |
| Toulouse | | 0.03 | 9.03 | 8.18 |
| Bordeaux | | | 0.05 | 7.18 |
| Sophia | | | | 0.06 |

| Throughput (Mb/s) | Orsay | Toulouse | Bordeaux | Sophia |
|---|---|---|---|---|
| Orsay | 890 | 78 | 90 | 102 |
| Toulouse | | 890 | 77 | 90 |
| Bordeaux | | | 890 | 83 |
| Sophia | | | | 890 |

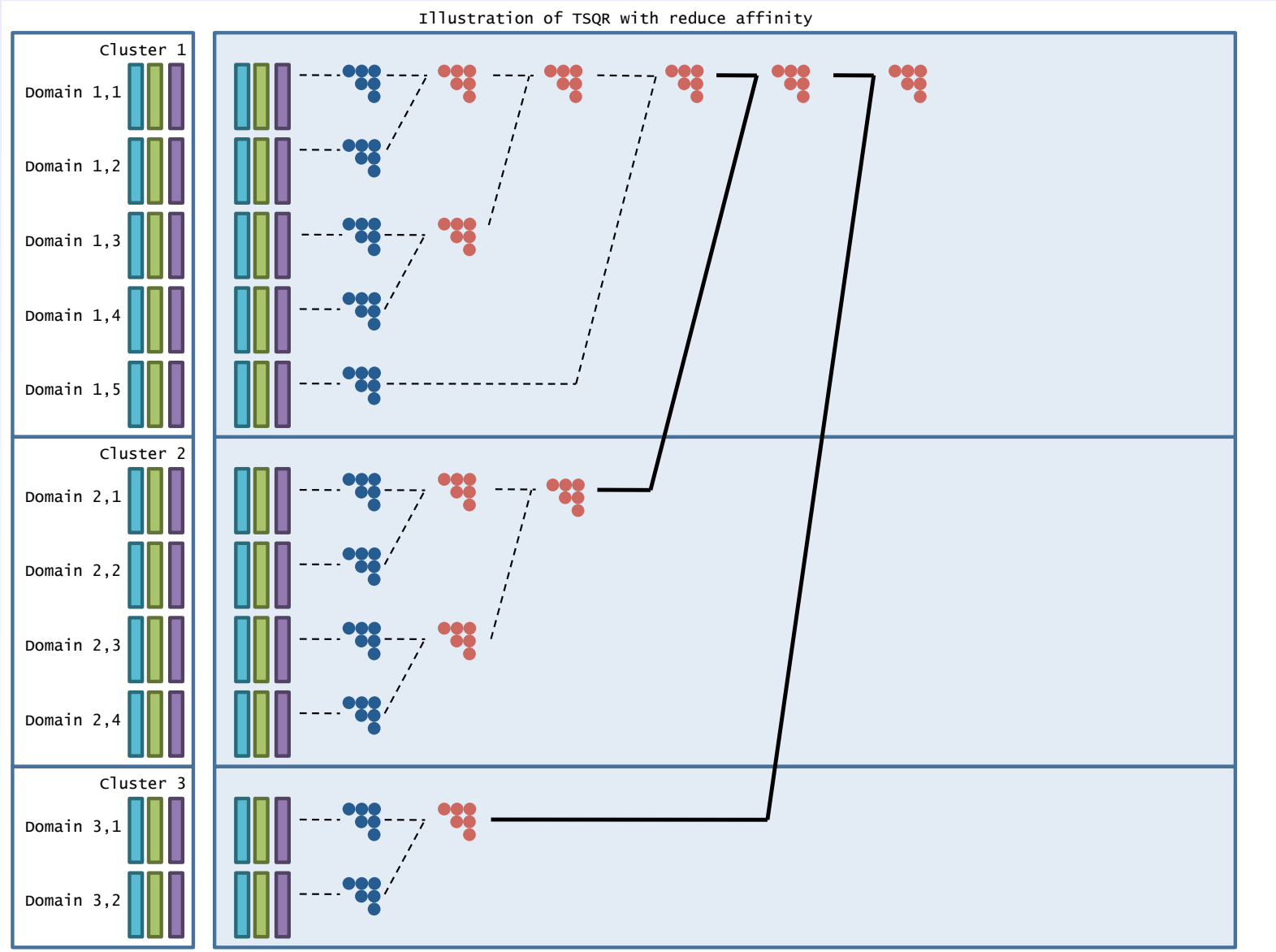
# 1D tile QR - binary tree



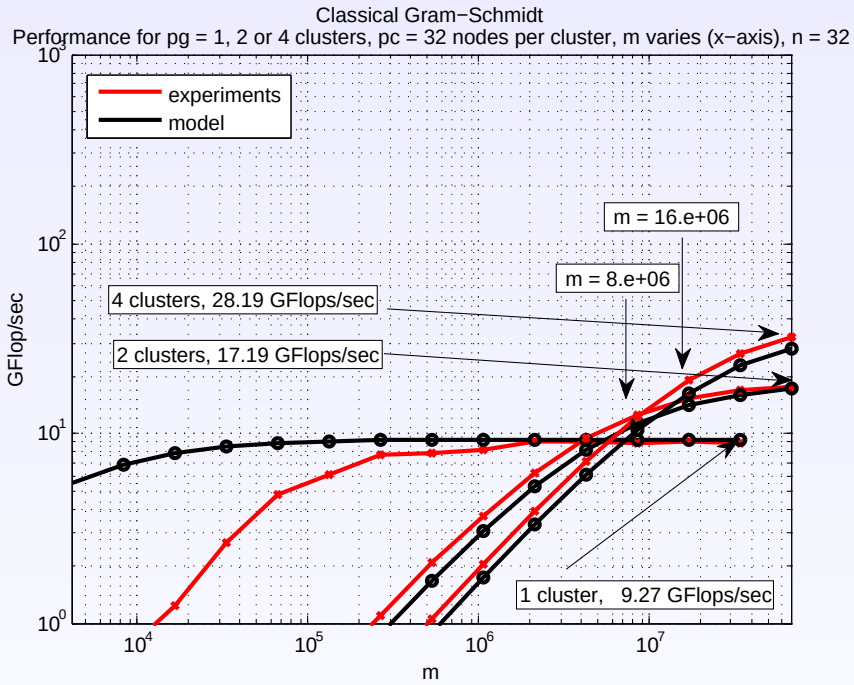
# 1D tile QR - binary tree

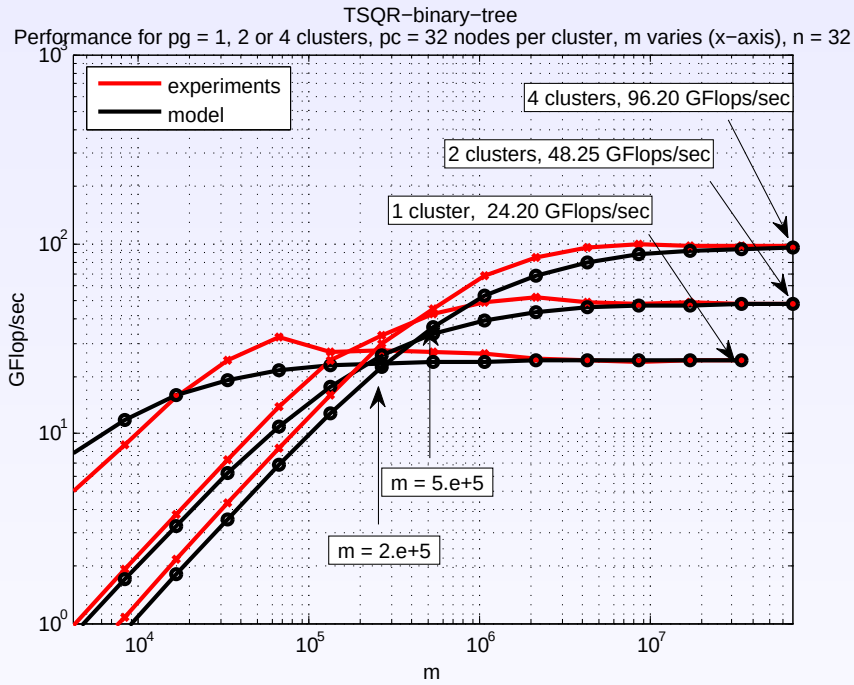


# 1D tile QR - binary tree



# 1D tile QR - binary tree



# 1D tile QR - binary tree

Consider
**architecture:** parallel case: $P$ processing units
**problem:** QR factorization of a $m$-by-$n$ TS matrix (TS = $m/P \geq n$)
**(main) assumption:** one processor has at least $mn/P$ part of the matrix and one processor performs at least $\frac{2mn^2}{P}$ flops

$\Rightarrow$ **1D tile QR algorithm with a binary tree**

**theory:**

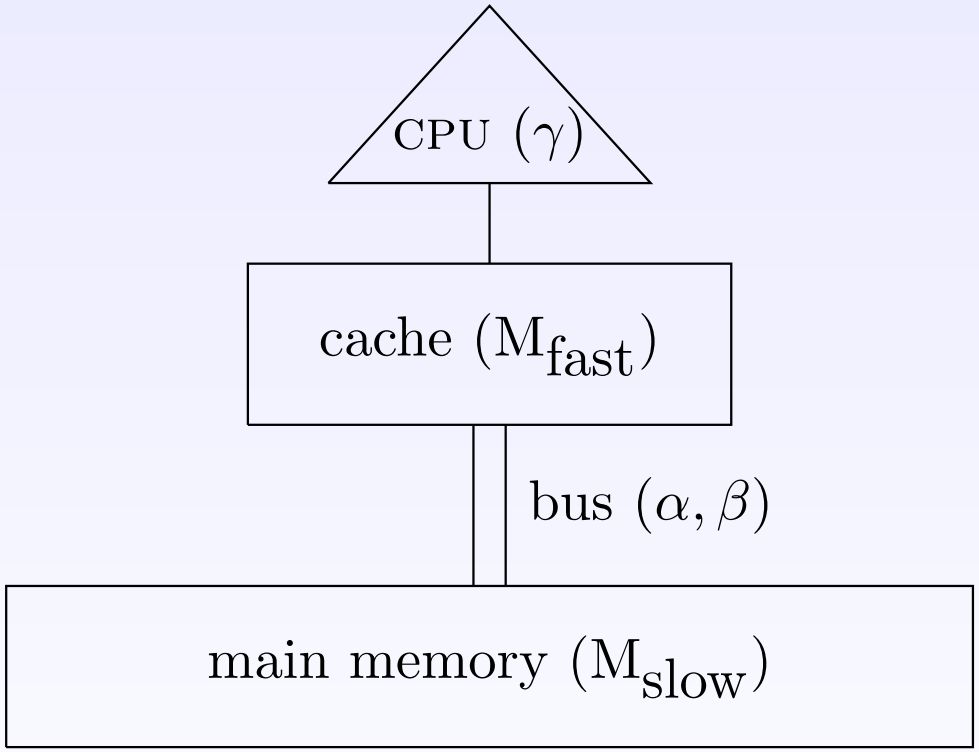
| | TSQR | ScaLAPACK-like | Lower bound |
|---|---|---|---|
| # flops | $\frac{2mn^2}{P} + \frac{2n^3}{3}\log P$ | $\frac{2mn^2}{P} - \frac{2n^3}{3P}$ | $\Theta\left(\frac{mn^2}{P}\right)$ |
| # words | $\frac{n^2}{2}\log P$ | $\frac{n^2}{2}\log P$ | $\frac{n^2}{2}\log P$ |
| **# messages** | $\log \mathbf{P}$ | $\mathbf{2n}\log \mathbf{P}$ | $\log \mathbf{P}$ |

# Communication for tile algorithms

- tall and skinny matrix, parallel distributed case

  $\Rightarrow$ **1D tile QR algorithm with a flat tree**
- tall and skinny matrix, sequential case
- general matrix, sequential case
- general matrix, parallel distributed case

# 1D tile QR - flat tree

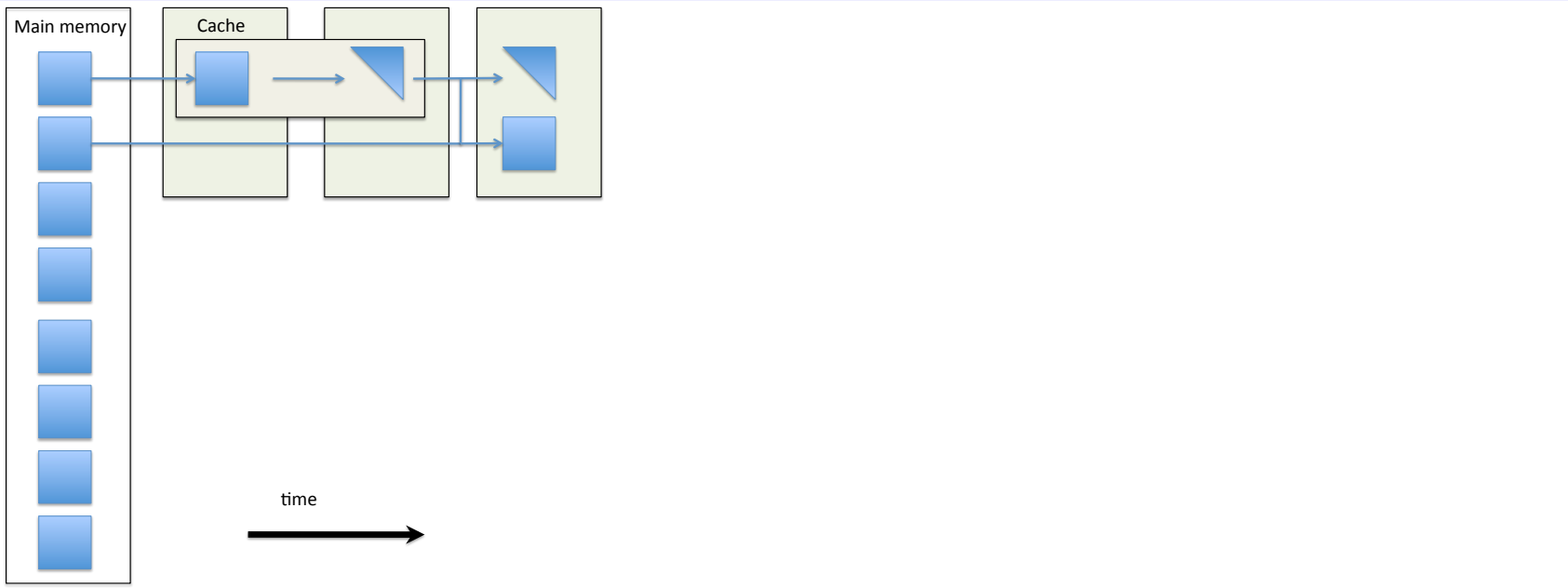


# 1D tile QR - flat tree

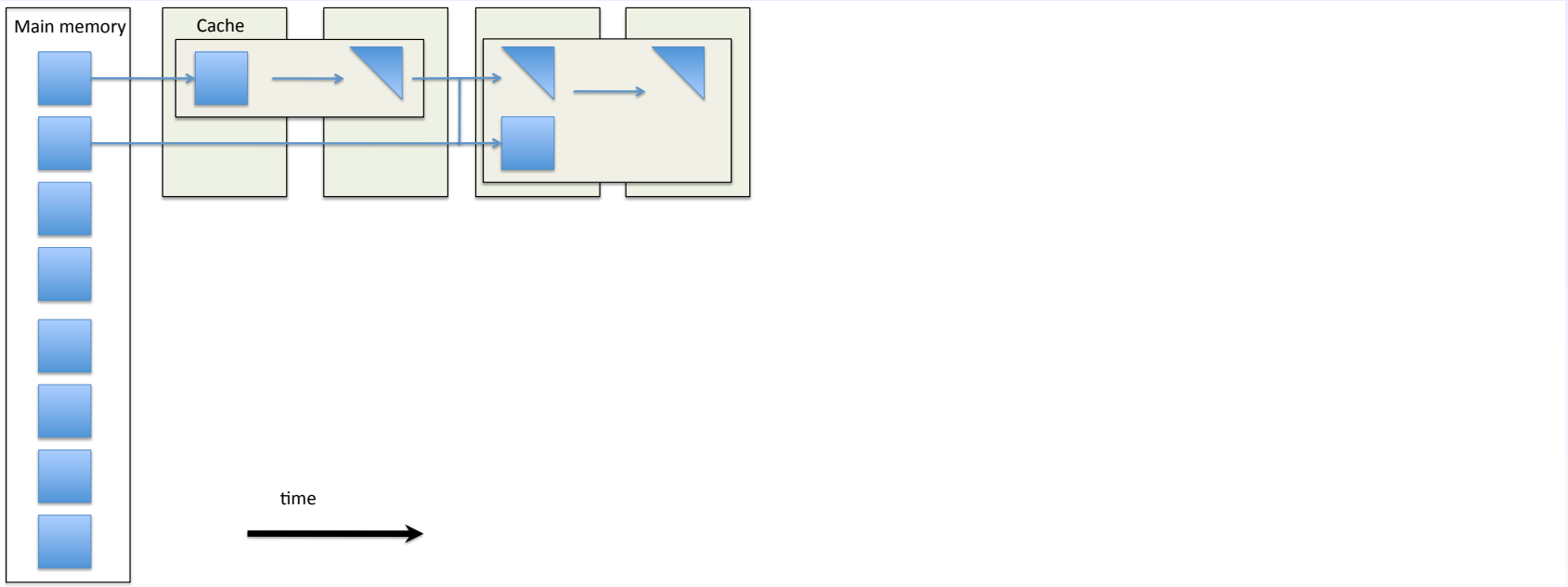
# 1D tile QR - flat tree

# 1D tile QR - flat tree

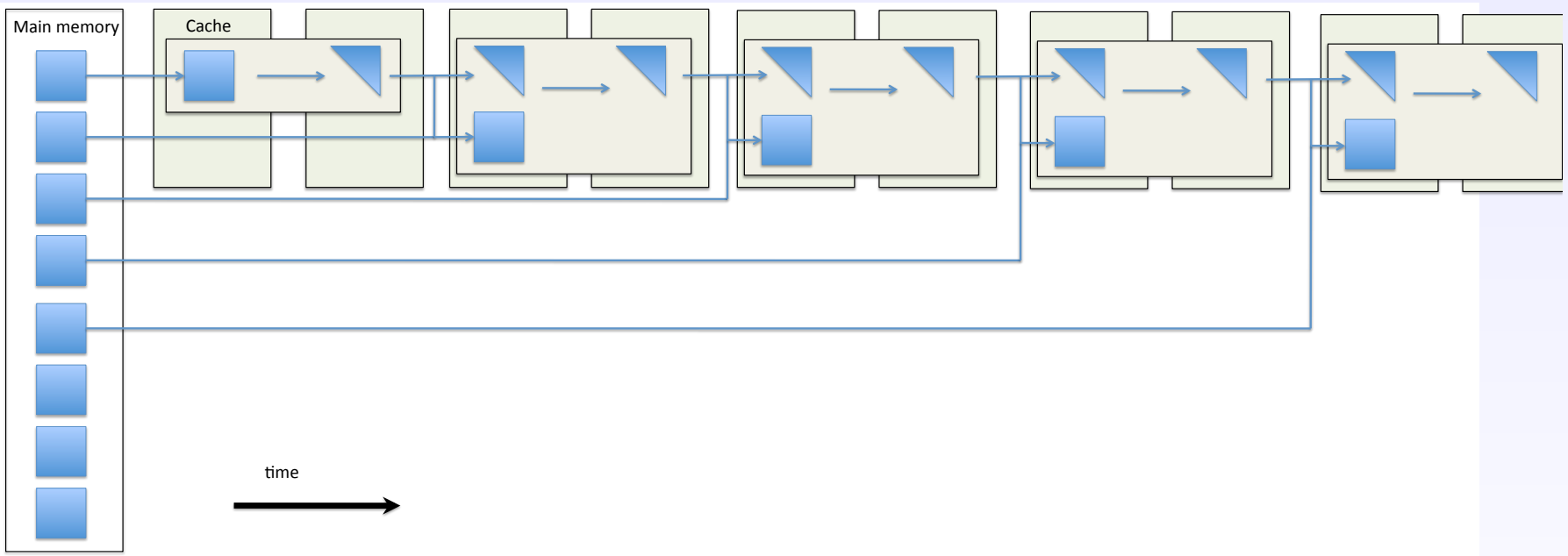
# 1D tile QR - flat tree



# 1D tile QR - flat tree



# 1D tile QR - flat tree



# 1D tile QR - flat tree

# 1D tile QR - flat tree

Consider
**architecture:** sequential case: one processing unit with cache of size ($W$)
**problem:** QR factorization of a $m$-by-$n$ TS matrix (TS = $m \geq n$ and $W \geq \frac{3}{2}n^2$)

$\Rightarrow$ **1D tile QR algorithm with a flat tree**

**theory:**

| | flat tree | LAPACK-like | Lower bound |
|---|---|---|---|
| # flops | $2mn^2$ | $2mn^2$ | $\Theta(mn^2)$ |
| # words | $mn$ | $\frac{m^2n^2}{2W}$ | $mn$ |
| # messages | $\frac{mn}{W}$ | $\frac{mn^2}{2W}$ | $\frac{mn}{W}$ |

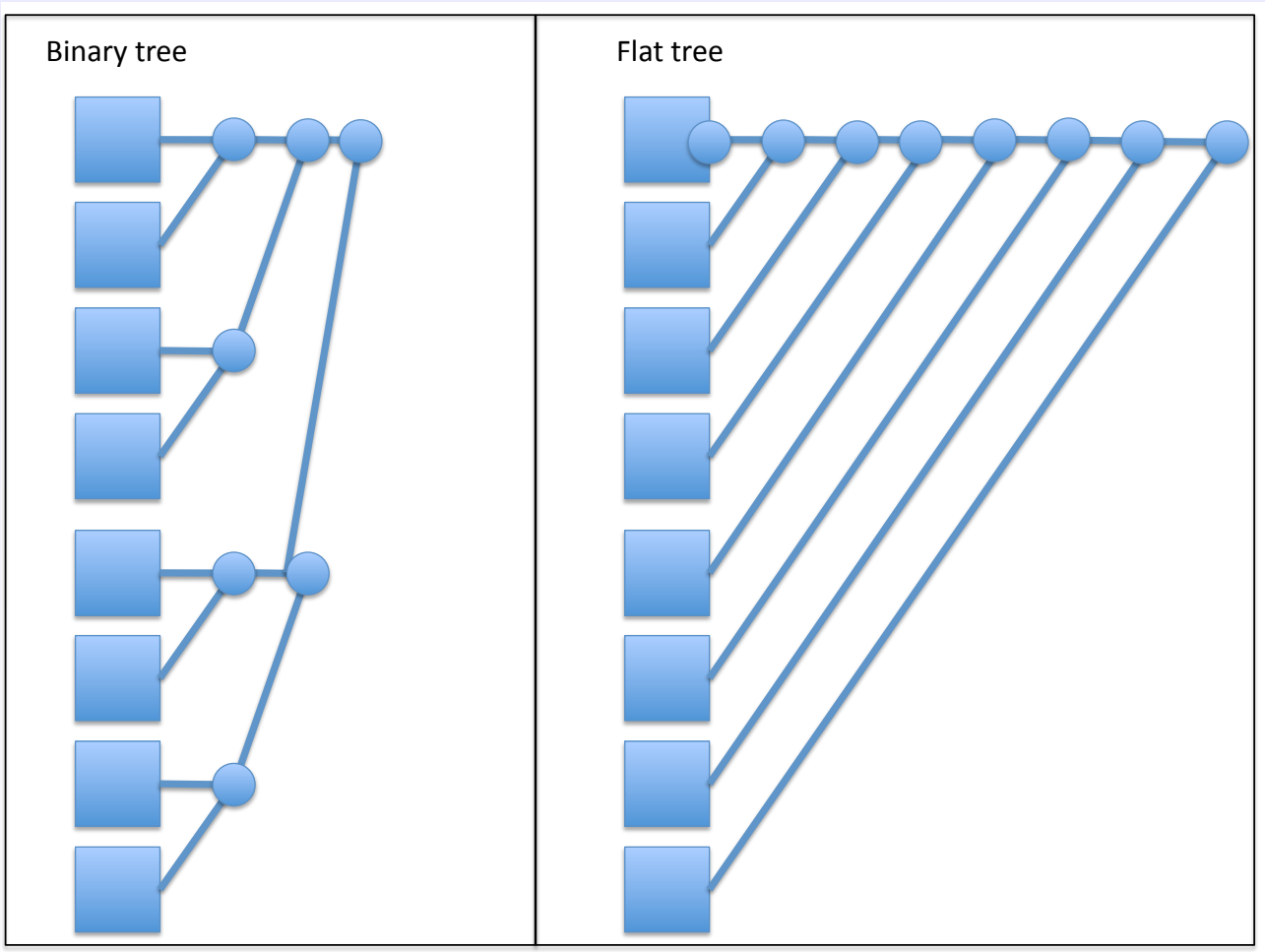
# Communication for tile algorithms

- tall and skinny matrix, parallel distributed case

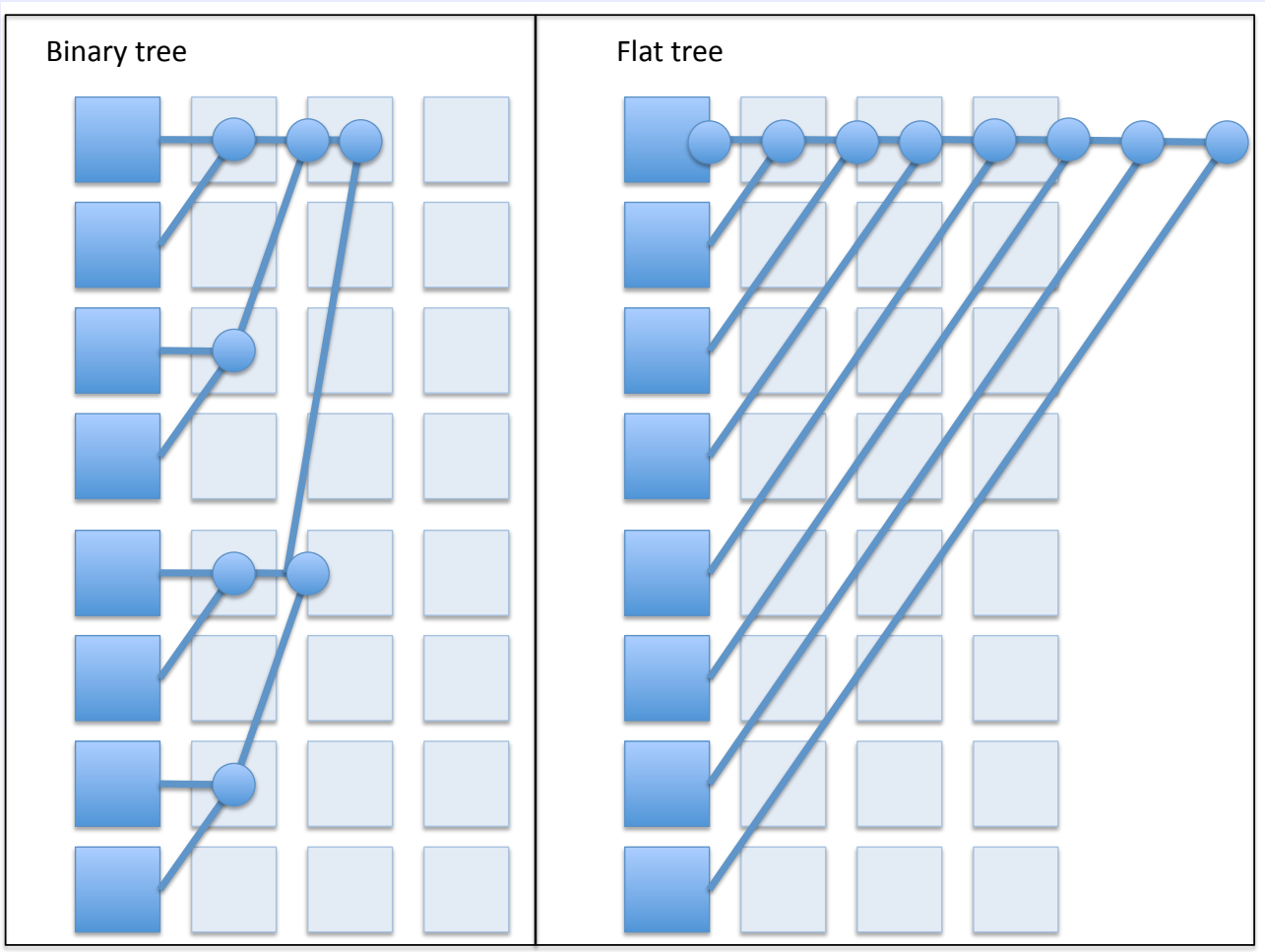
  $\Rightarrow$ **1D tile QR algorithm with a flat tree**
- tall and skinny matrix, sequential case

  $\Rightarrow$ **1D tile QR algorithm with a flat tree**
- general matrix, sequential case
- general matrix, parallel distributed case

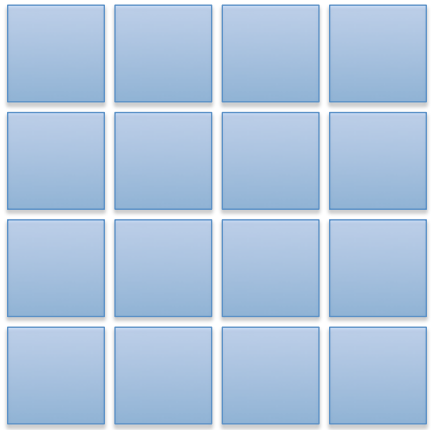
# 2D tile QR



# 2D tile QR

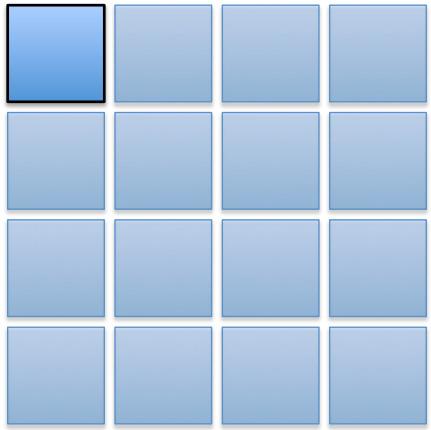


# 2D tile QR - flat tree

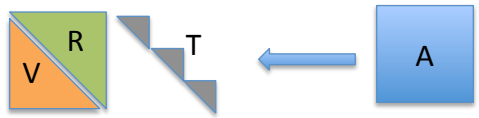


```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

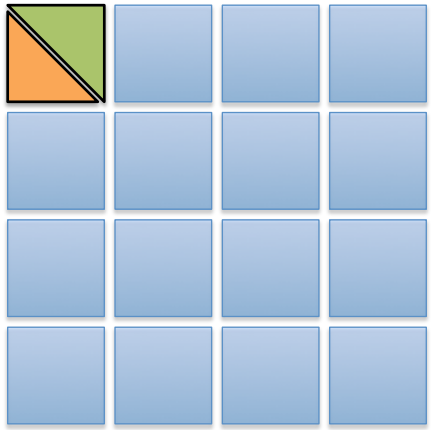


**1. dgeqrt**(A[0][0], T[0][0]);

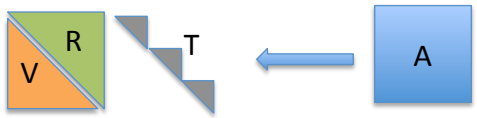



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

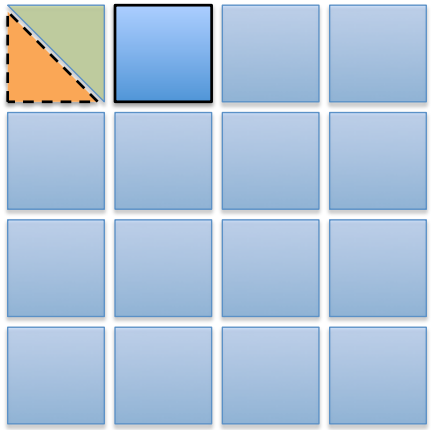


**1. dgeqrt**(A[0][0], T[0][0]);

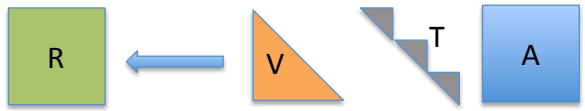



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

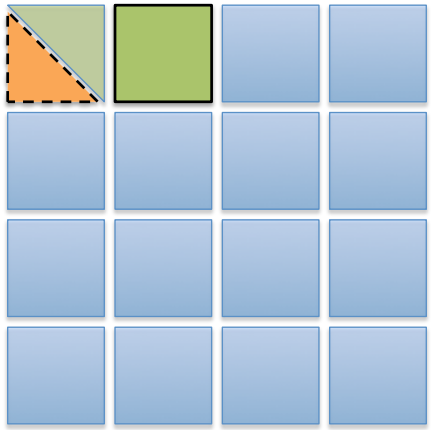


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);

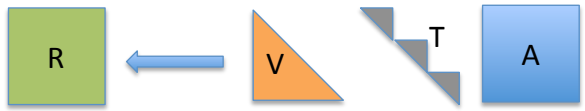



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

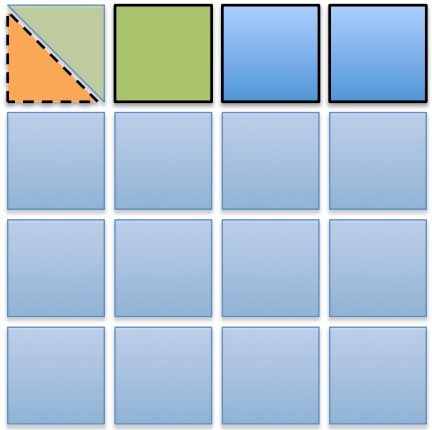


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);

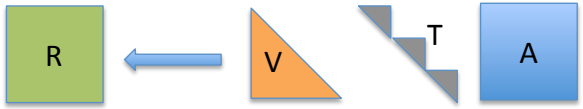



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

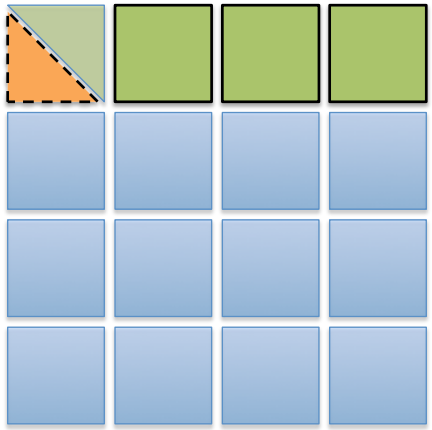


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);

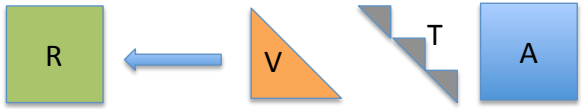



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

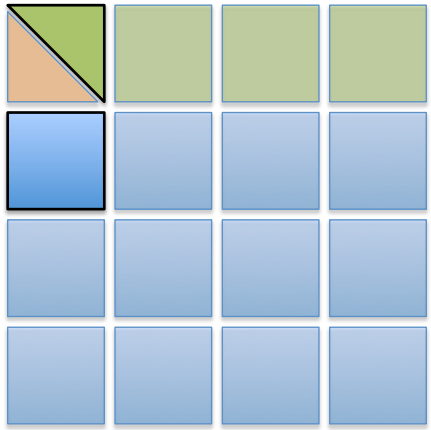


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);

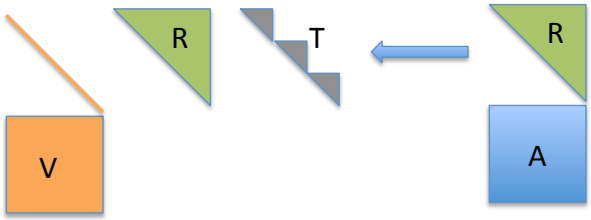



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

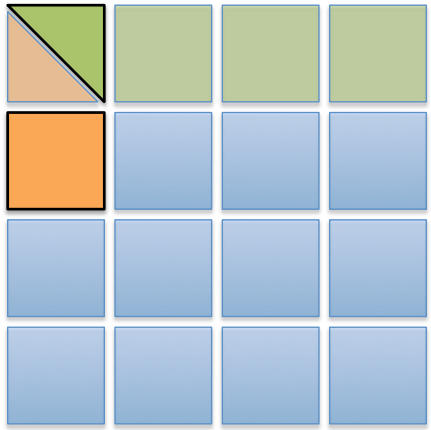


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);

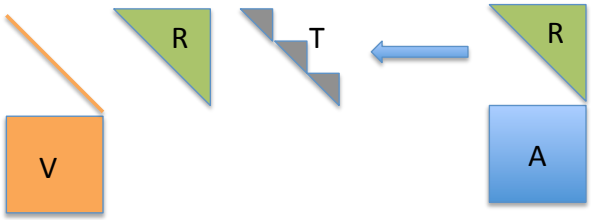



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

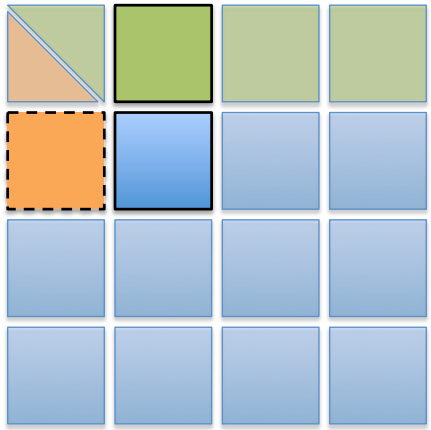


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);

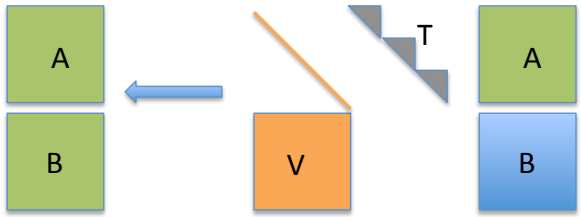



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

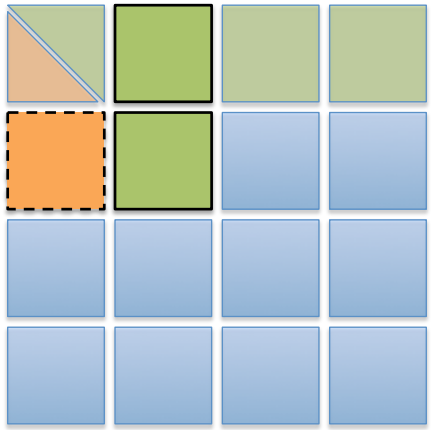


…
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

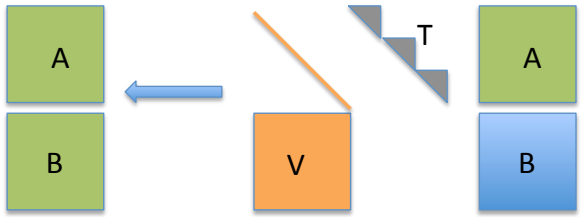



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree

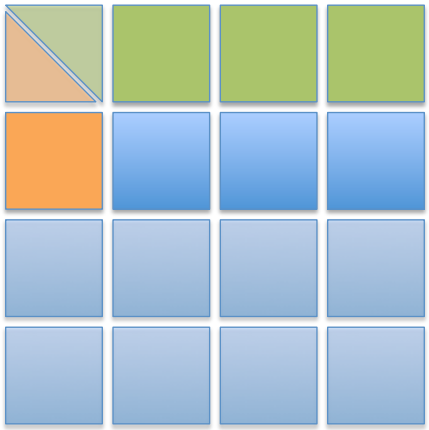


…
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

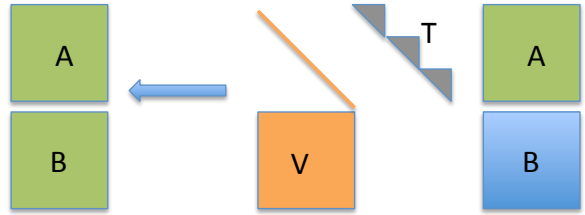



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# 2D tile QR - flat tree



...
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
7. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
8. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);




```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
        }
}
```

# Lower bounds

1. Extends previous lower bounds on the volume of communication for matrix-matrix multiplication from
   - Hong and Kung (81) - sequential case: $\Omega(\frac{n^3}{\sqrt{M}})$
   - Irony,Toledo,Tiskin (04) - parallel case: $\Omega(\frac{n^2}{\sqrt{P}})$
2. For LU, observe that:

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix}$$

therefore lower bound for matrix-matrix multiply (latency, bandwidth and operations) also holds for LU.

# Lower bounds

1. Extends previous lower bounds on the volume of communication for matrix-matrix multiplication from
   - Hong and Kung (81) - sequential case: $\Omega(\frac{n^3}{\sqrt{M}})$
   - Irony,Toledo,Tiskin (04) - parallel case: $\Omega(\frac{n^2}{\sqrt{P}})$
2. For LU, observe that:

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix}$$

therefore lower bound for matrix-matrix multiply (latency, bandwidth and operations) also holds for LU.

# 2D tile QR - binary tree

| | 2D tile QR binary tree | | ScaLAPACK Algorithm PDGEQRF | | Lower bound |
|---|---|---|---|---|---|
| # flops | $\left(\frac{4}{3}\right) \cdot (n^3/P)$ | ✓ | $\left(\frac{4}{3}\right) \cdot (n^3/P)$ | ✓ | $\mathcal{O}\left(n^3/P\right)$ |
| # words | $\left(\frac{3}{4} \cdot \log P\right) \cdot \left(n^2/\sqrt{P}\right)$ | ✓ | $\left(\frac{3}{4} \cdot \log P\right) \cdot \left(n^2/\sqrt{P}\right)$ | ✓ | $\mathcal{O}\left(n^2/\sqrt{P}\right)$ |
| # messages | $\left(\frac{3}{8} \cdot \log^3 P\right) \cdot \left(\sqrt{P}\right)$ | ✓ | $\left(\frac{5}{4} \cdot \log^2 P\right) \cdot (n)$ | × | $\mathcal{O}\left(\sqrt{P}\right)$ |
| | 2D tile QR binary tree | ✓ | ~~ScaLAPACK~~ | × | |

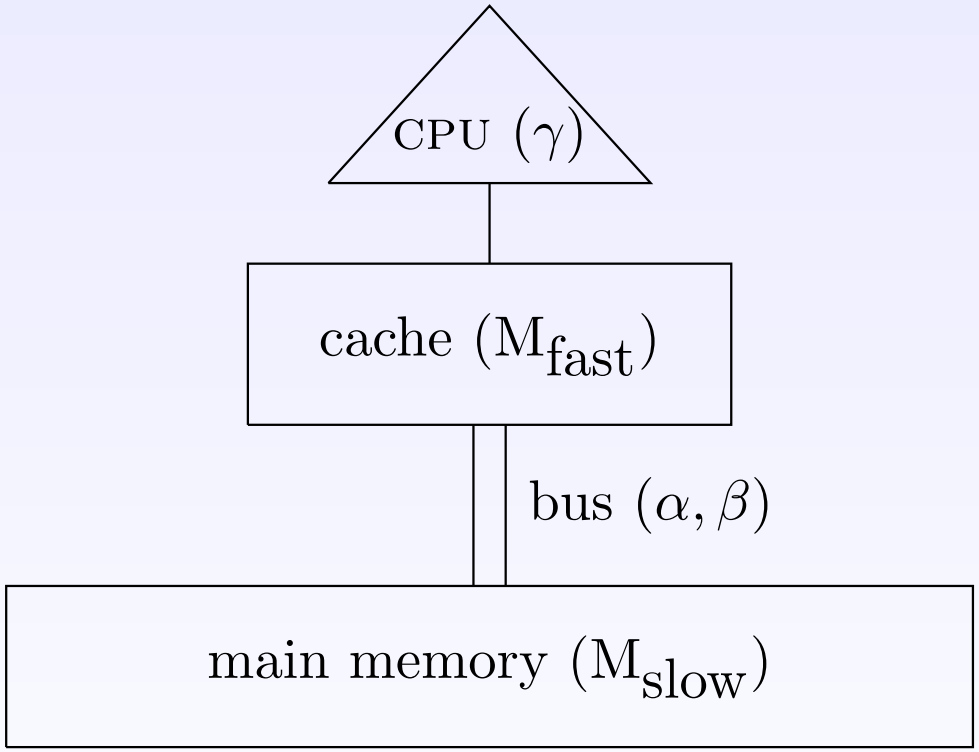
Performance models of parallel CAQR and ScaLAPACK's parallel QR factorization PDGEQRF on a square $n \times n$ matrix with $P$ processors, along with lower bounds on the number of flops, words, and messages. The matrix is stored in a 2-D $P_r \times P_c$ block cyclic layout with square $b \times b$ blocks. We choose $b$, $P_r$, and $P_c$ optimally and independently for each algorithm. Everything (messages, words, and flops) is counted along the critical path.

# 2D tile QR - flat tree

| | 2D tile QR flat tree | | LAPACK Algorithm DGEQRF | | Lower bound |
|---|---|---|---|---|---|
| # flops | $\left(\frac{4}{3}\right) \cdot \left(n^3\right)$ | ✓ | $\left(\frac{4}{3}\right) \cdot \left(n^3\right)$ | ✓ | $\mathcal{O}\left(n^3\right)$ |
| # words | $3 \cdot \frac{n^3}{\sqrt{W}}$ | ✓ | $\frac{1}{3} \cdot \frac{n^4}{W}$ | ✓ | $\mathcal{O}\left(\frac{n^3}{\sqrt{W}}\right)$ |
| # messages | $12 \cdot \frac{n^3}{W^{3/2}}$ | ✓ | $\frac{1}{2} \cdot \frac{n^3}{W}$ | × | $\mathcal{O}\left(\frac{n^3}{W^{3/2}}\right)$ |
| | 2D tile QR flat tree | ✓ | ~~LAPACK~~ | × | |

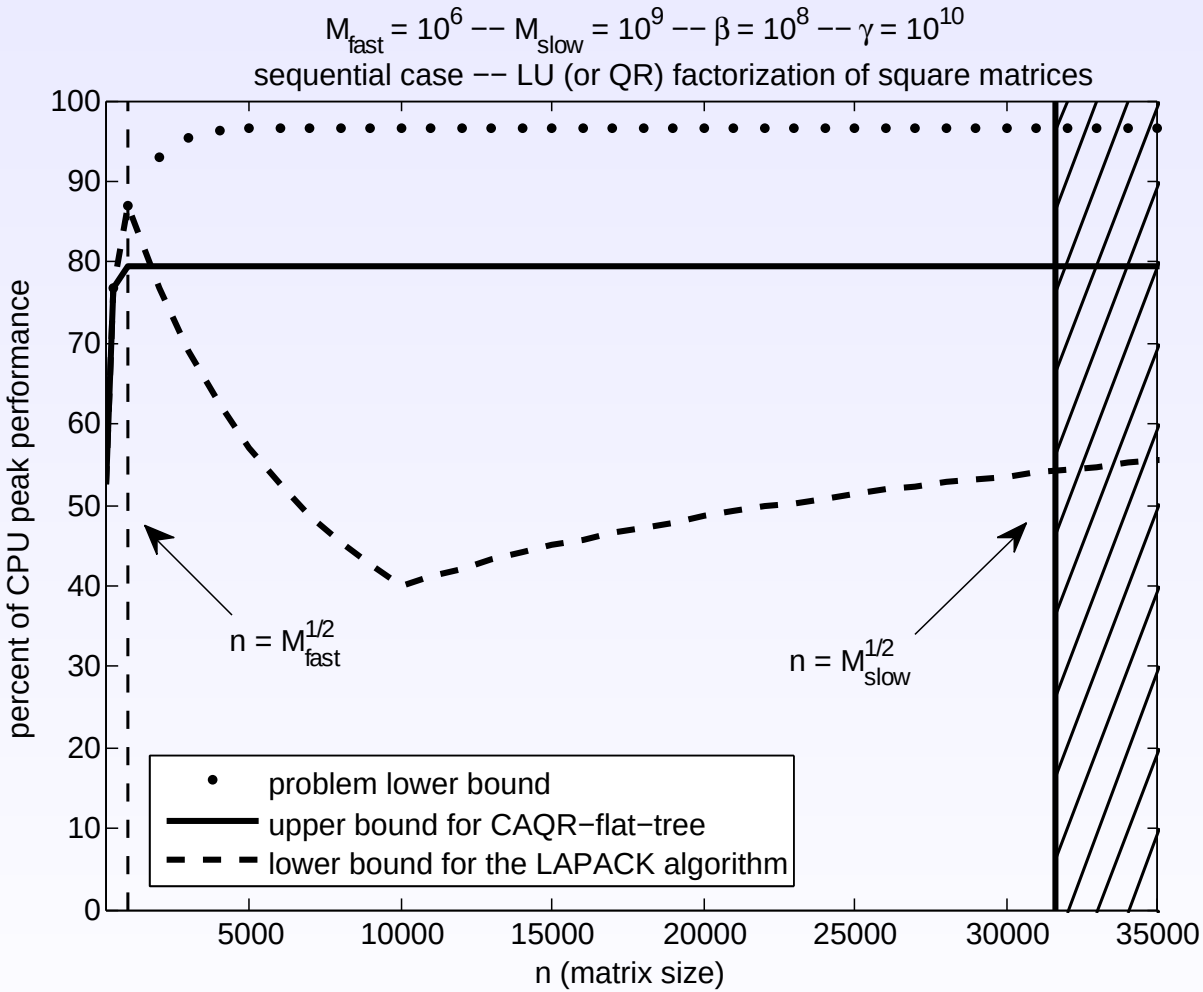
Performance models of sequential CAQR and blocked sequential Householder QR on a square $n \times n$ matrix with fast memory size $W$, along with lower bounds on the number of flops, words, and messages.

# 2D tile QR - flat tree

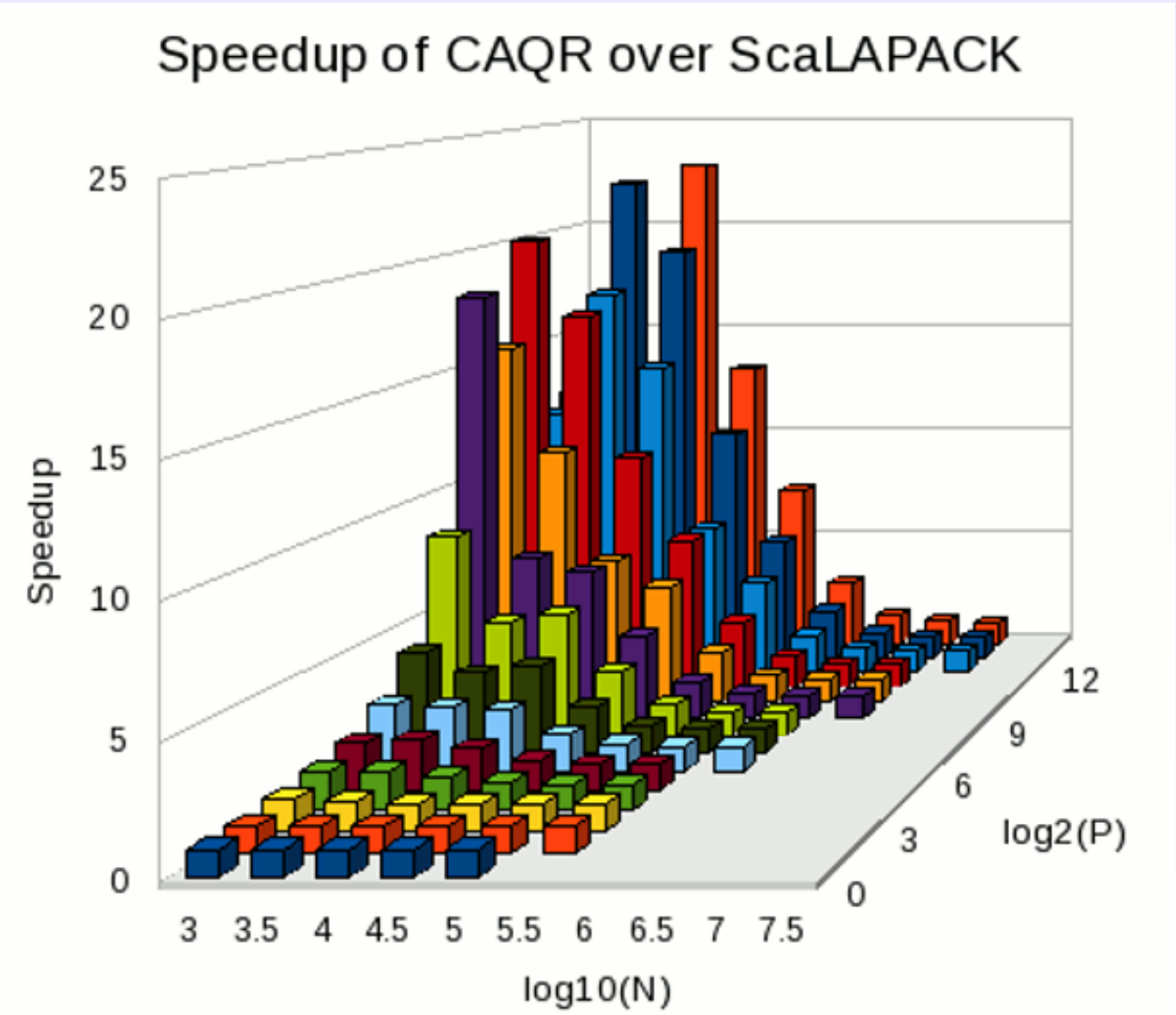



$M_{fast} = 1$ (MB)
$M_{slow} = 1$ (GB)
$\alpha = 0$ (SEC)
$\beta = 10^8$ (GB/SEC)
$\gamma = 10^{10}$ (GFLOPS/SEC)

# 2D tile QR - flat tree



# 2D tile QR - binary tree



# Communication for tile algorithms

- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Communication for tile algorithms

- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Communication for tile algorithms

- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Communication for tile algorithms

- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Communication for tile algorithms

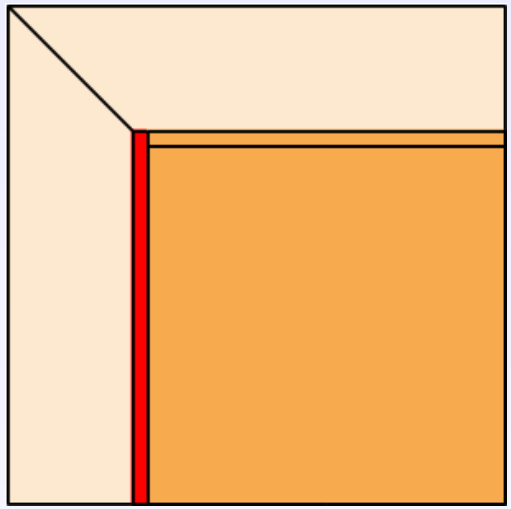
- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Communication for tile algorithms

- Tile algorithms enable to reduce communication in the sequential case, and in the parallel case wrt existing software
- We can derive communication lower bounds for our problems
- We can attain (polylogarithmically) (assymptotically) communication lower bounds with tile algorithms
- Any tree is possible for the panel factorization
- We (Julien) used tile algorithms to minimize communication,
- We (Emmanuel) now explain them in the context of maximizing parallelism

# Software/Algorithms follow hardware evolution in time

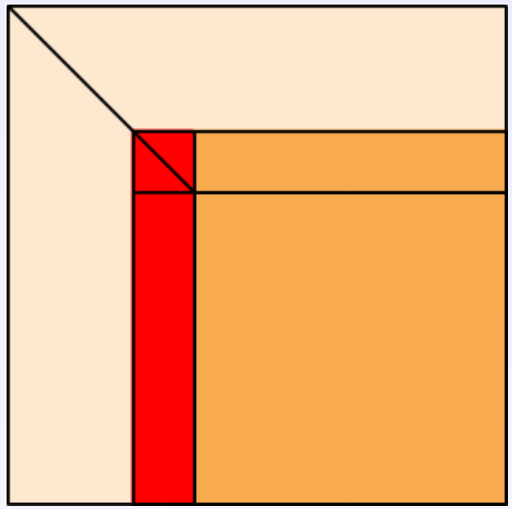
- ★ 70's - LINPACK, vector operations:
  Level-1 BLAS operation
- ★ 80's - LAPACK, block, cache-friendly:
  Level-3 BLAS operation
- ★ 90's - ScaLAPACK, distributed memory:
  PBLAS Message passing



Video: LAPACK 1 Thread

# Software/Algorithms follow hardware evolution in time

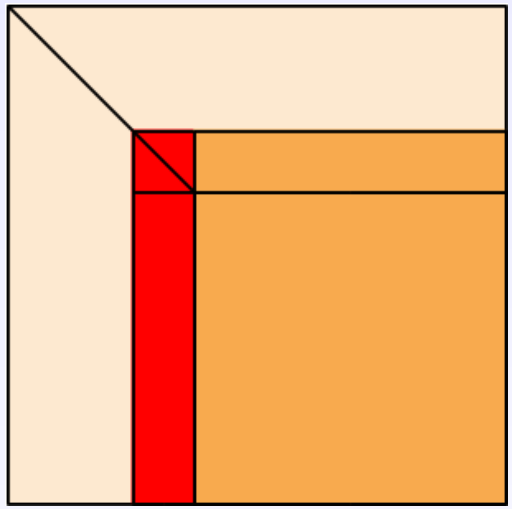
- ★ 70's - LINPACK, vector operations:
  Level-1 BLAS operation
- ★ 80's - LAPACK, block, cache-friendly:
  Level-3 BLAS operation
- ★ 90's - ScaLAPACK, distributed memory:
  PBLAS Message passing



Video: LAPACK 1 Thread

# Software/Algorithms follow hardware evolution in time

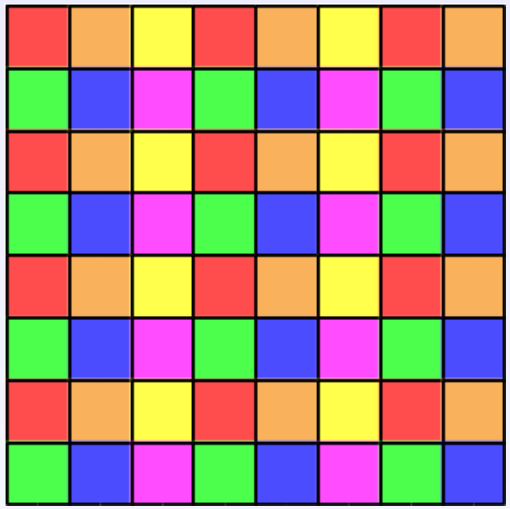
- 70's - LINPACK, vector operations:
  Level-1 BLAS operation
- 80's - LAPACK, block, cache-friendly:
  Level-3 BLAS operation
- 90's - ScaLAPACK, distributed memory:
  PBLAS Message passing



Video: LAPACK 1 Thread

# Software/Algorithms follow hardware evolution in time

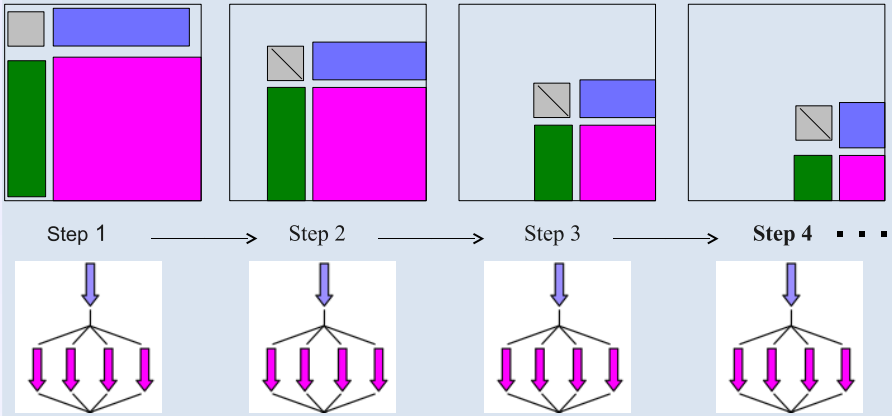
- 70's - LINPACK, vector operations:
  Level-1 BLAS operation
- 80's - LAPACK, block, cache-friendly:
  Level-3 BLAS operation
- 90's - ScaLAPACK, distributed memory:
  PBLAS Message passing



Video: LAPACK 1 Thread

# LAPACK QR factorization

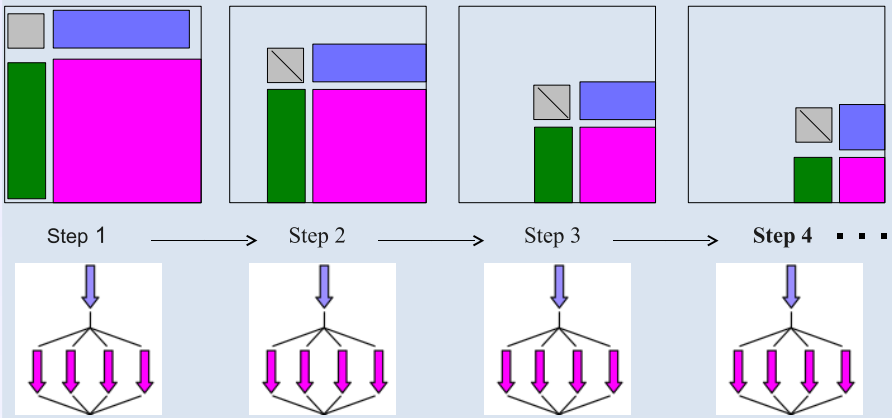
## LAPACK QR factorization




- ★ Block algorithm
- → (Movie L1)
- ★ Fork-join parallelism
- ★ Multithreaded BLAS
- → (Movie L4);

# LAPACK QR factorization

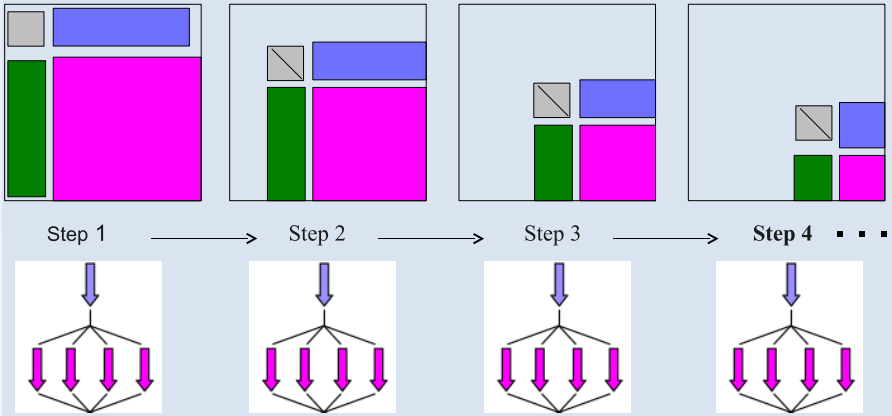
## LAPACK QR factorization




- ⋆ Block algorithm
- → (Movie L1)
- ⋆ Fork-join parallelism
- ⋆ Multithreaded BLAS
- → (Movie L4);

# LAPACK QR factorization

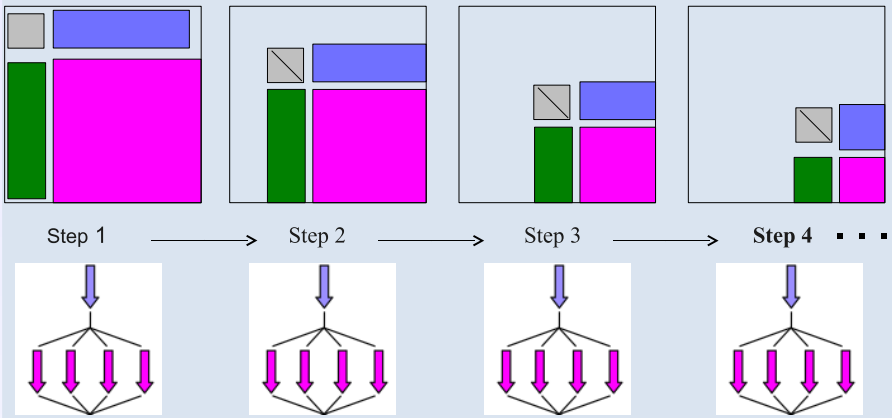
## LAPACK QR factorization




- ★ Block algorithm
  → (Movie L1)
- ★ Fork-join parallelism
- ★ Multithreaded BLAS
  → (Movie L4);

# LAPACK QR factorization

## LAPACK QR factorization




- ★ Block algorithm

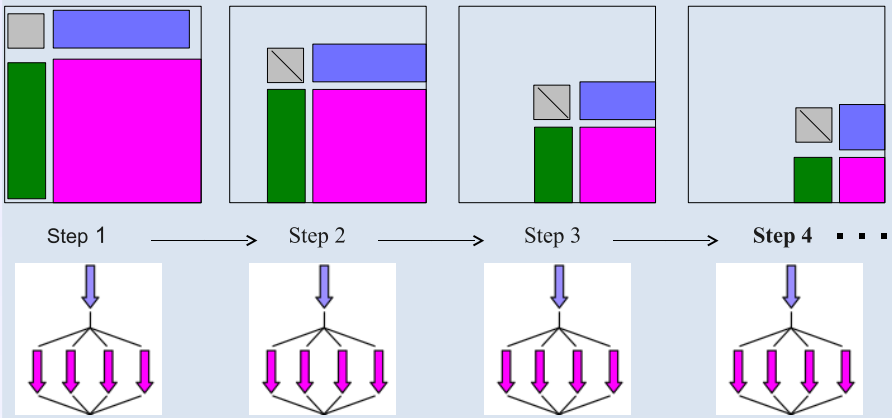
→ (Movie L1)

- ★ Fork-join parallelism
- ★ Multithreaded BLAS

→ (Movie L4);

# LAPACK QR factorization

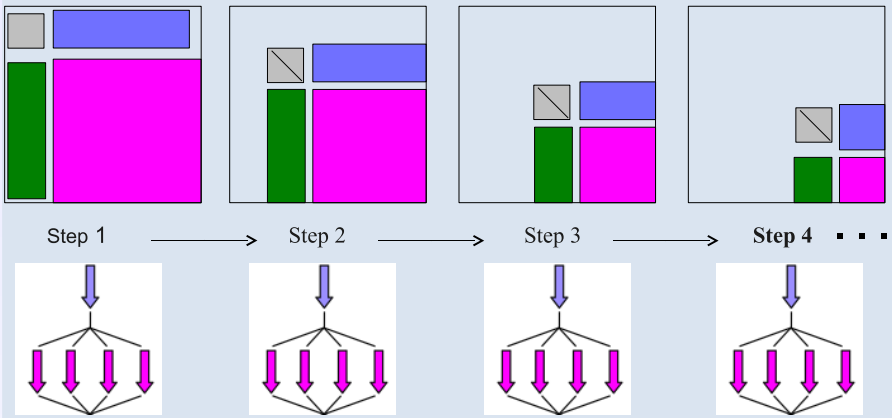
## LAPACK QR factorization



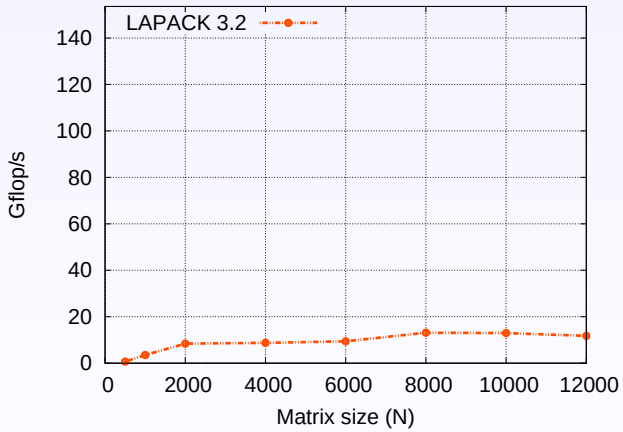

- ★ Block algorithm
- → (Movie L1)
- ★ Fork-join parallelism
- ★ Multithreaded BLAS
- → (Movie L4);

# LAPACK QR factorization

## LAPACK QR factorization



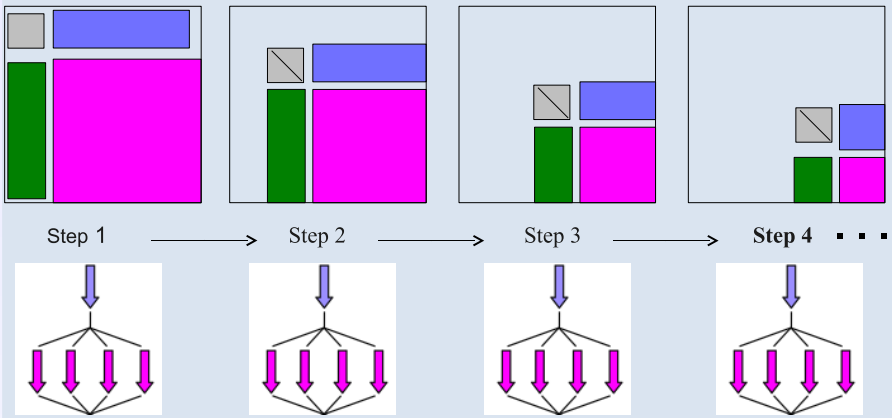

- ★ Block algorithm
  → (Movie L1)
- ★ Fork-join parallelism
- ★ Multithreaded BLAS
  → (Movie L4);




Intel Xeon E7340 quad-socket quad-core
(16 cores total)

# LAPACK QR factorization

## LAPACK QR factorization




- ★ Block algorithm

→ (Movie L1)

- ★ Fork-join parallelism
- ★ Multithreaded BLAS

→ (Movie L4);

Need for tile algorithms !

# Three-layers paradigm

**High-level algorithm**

2D Tile QR - flat tree

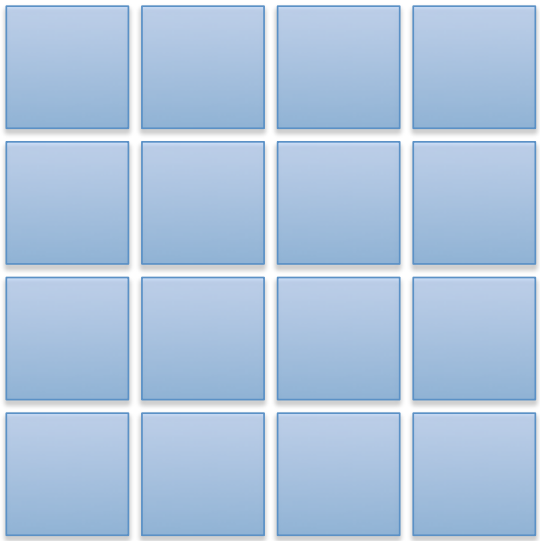
**Runtime System**

**Device kernels**

**CPU core**

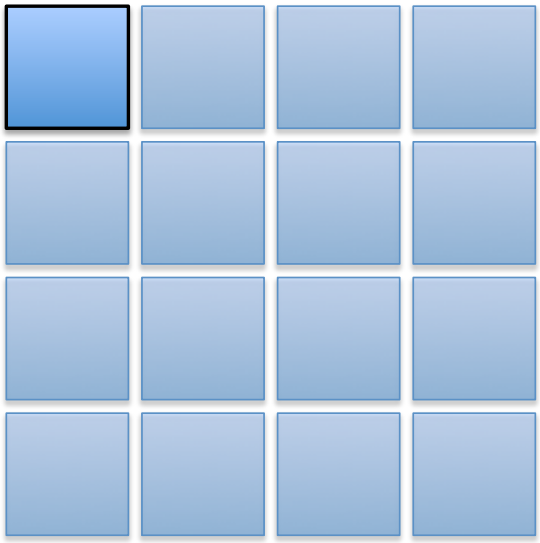
**GPU**

# DAG - 2D tile QR - flat tree

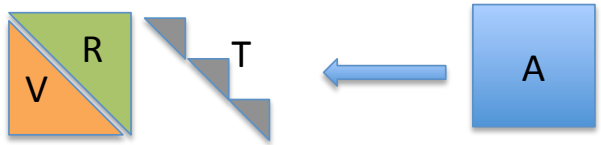


```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



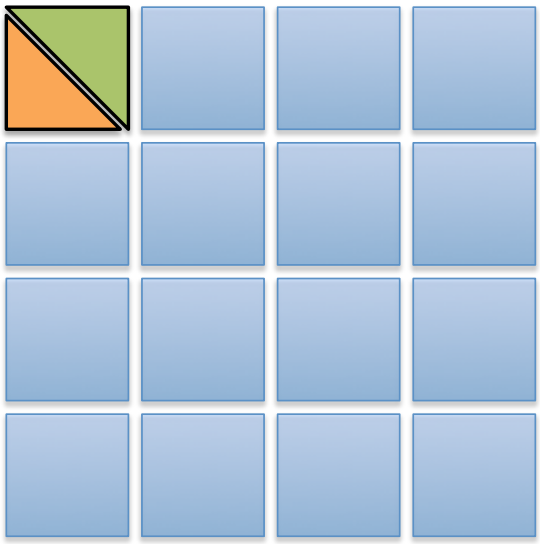
**1. dgeqrt**(A[0][0], T[0][0]);



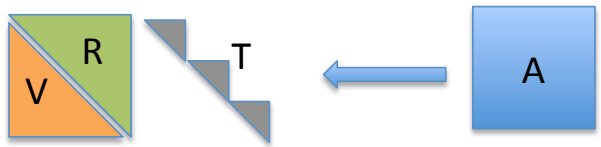


```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



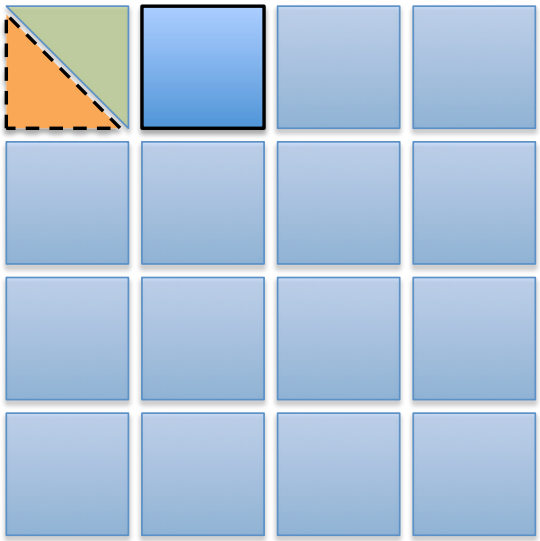
**1. dgeqrt**(A[0][0], T[0][0]);



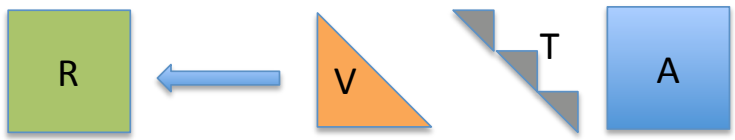


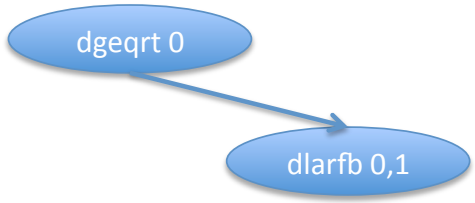
```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



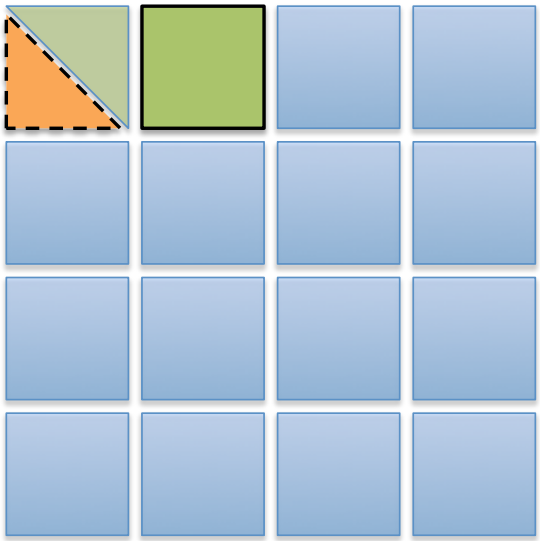
1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);



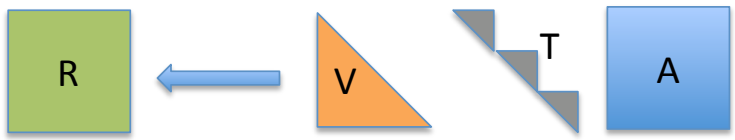



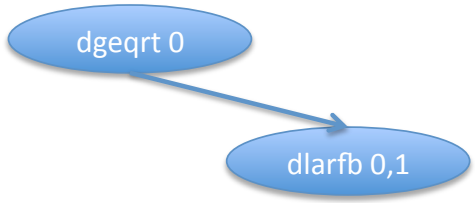

```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



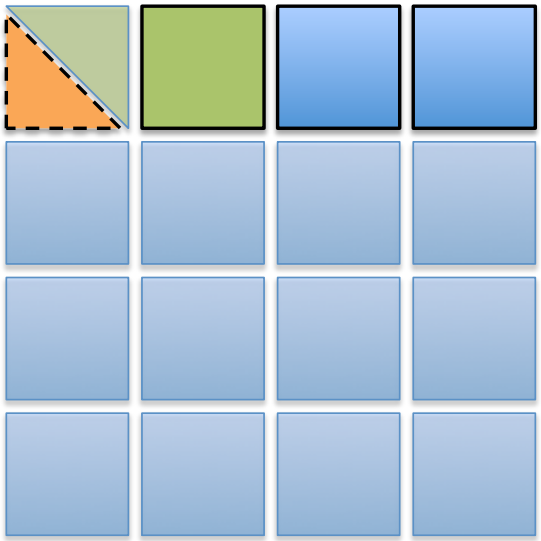
1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);



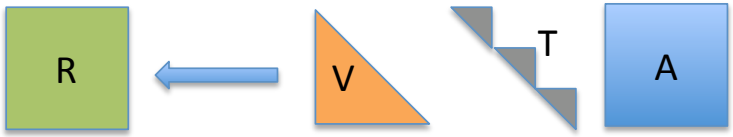



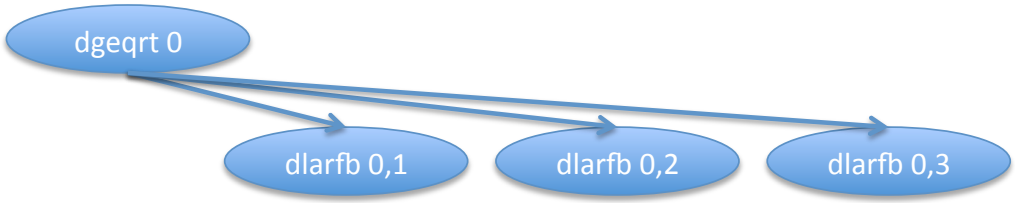

```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



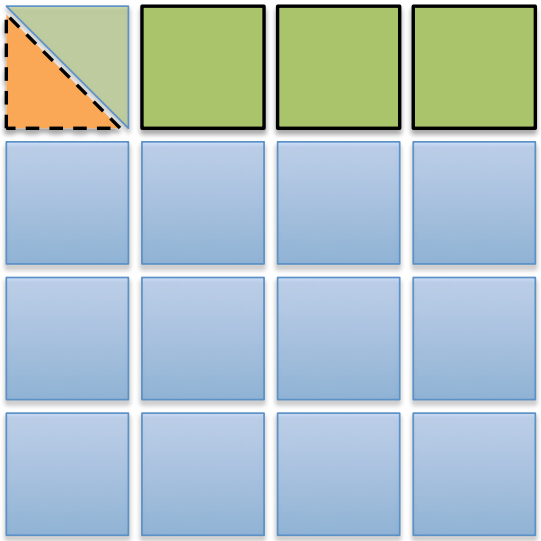
1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);



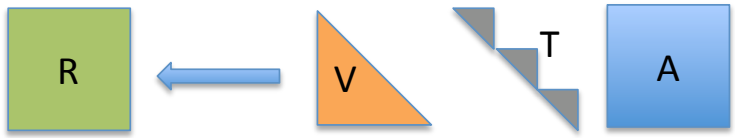



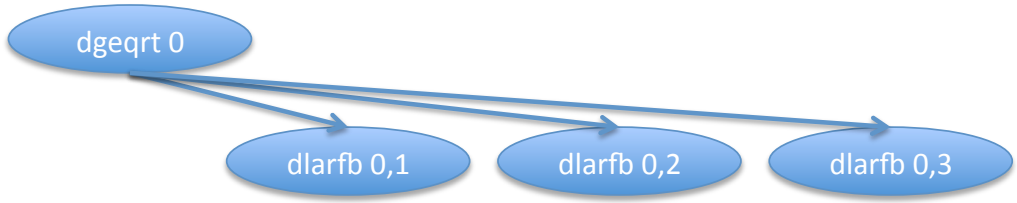

```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



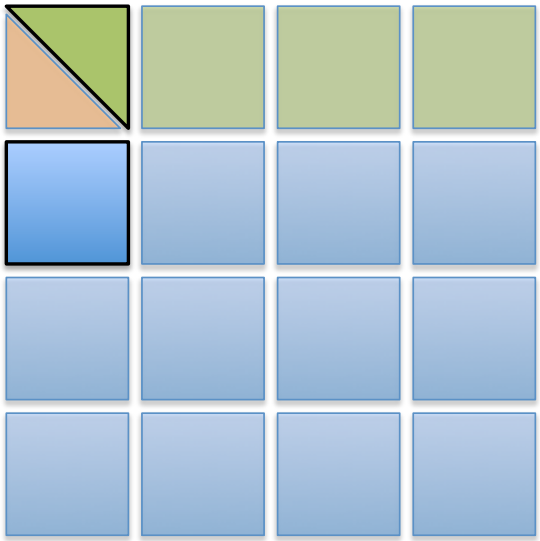
1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);



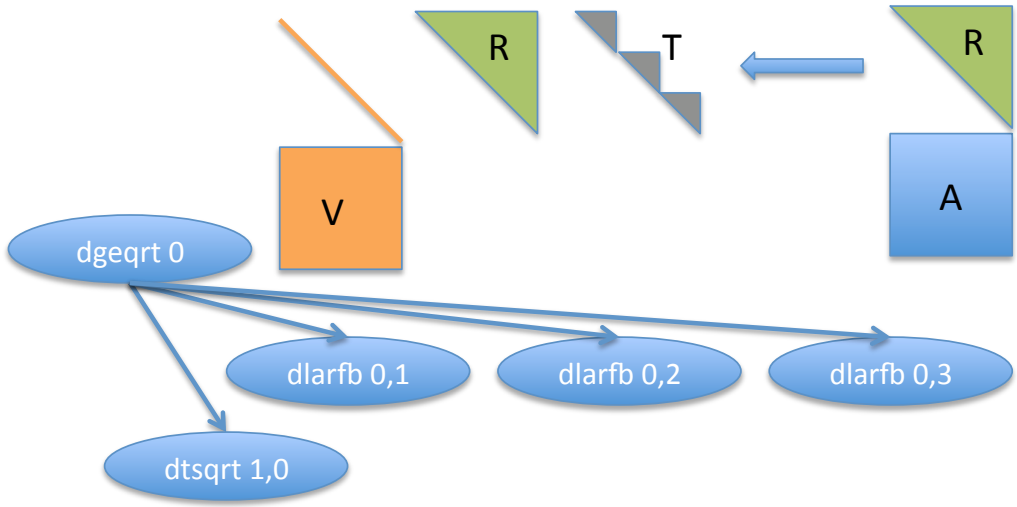




```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree

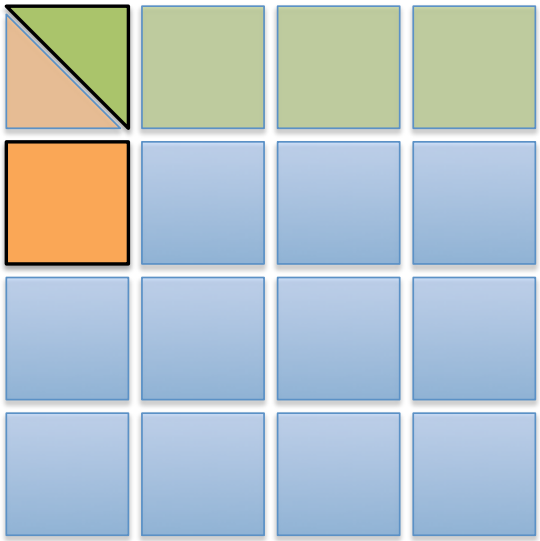


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);

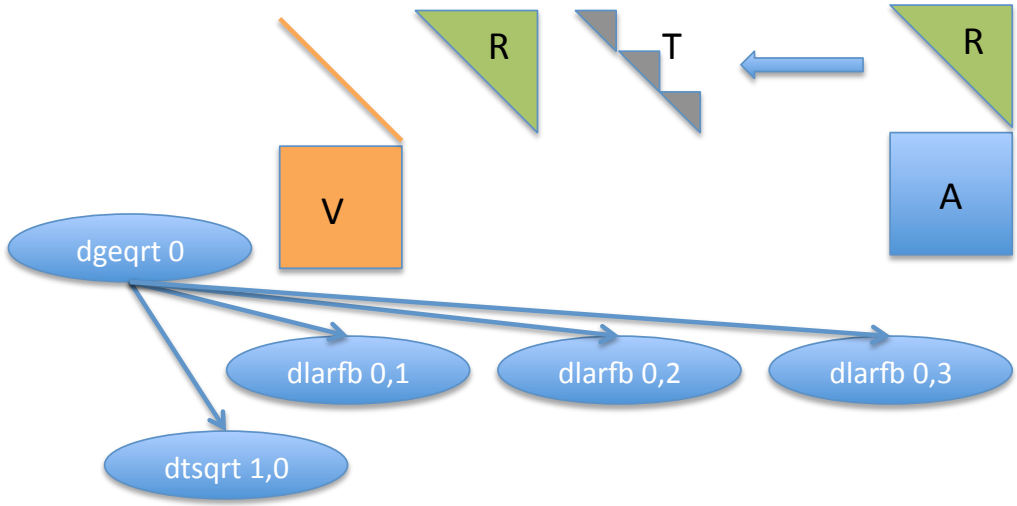



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree

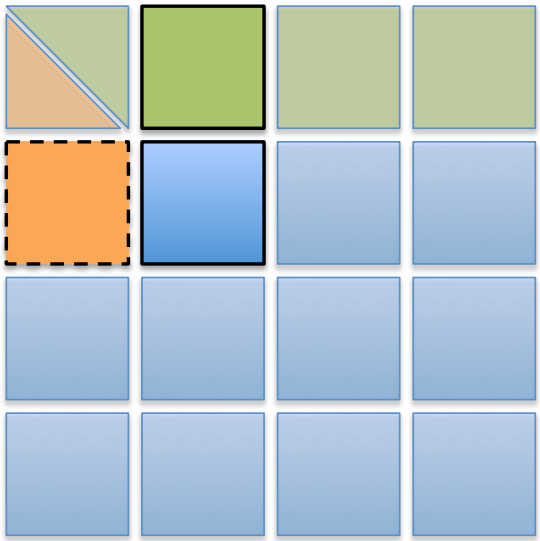


1. **dgeqrt**(A[0][0], T[0][0]);
2. **dlarfb**(A[0][0], T[0][0], A[0][1]);
3. **dlarfb**(A[0][0], T[0][0], A[0][2]);
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);



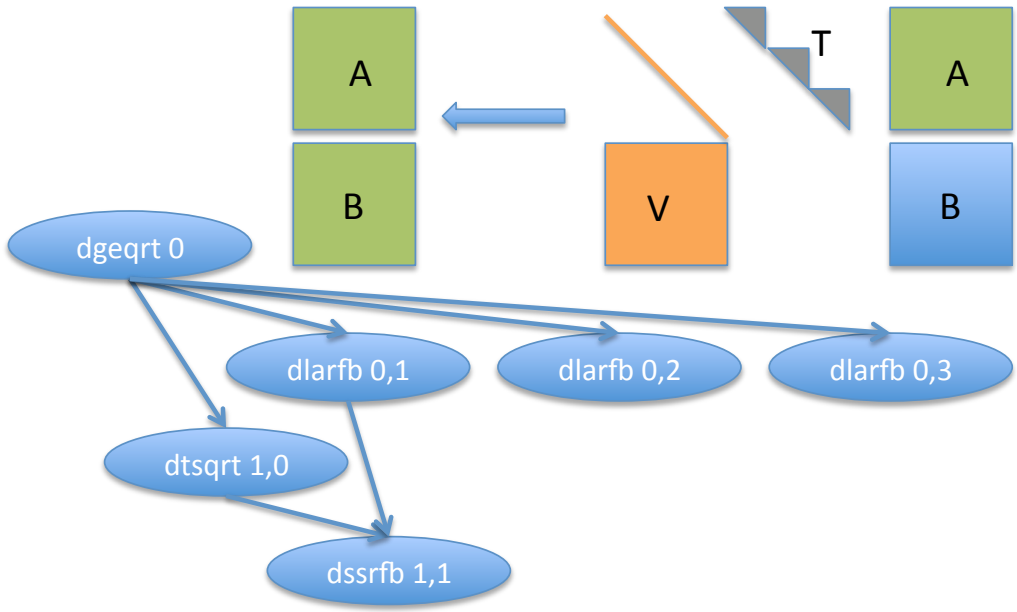

```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



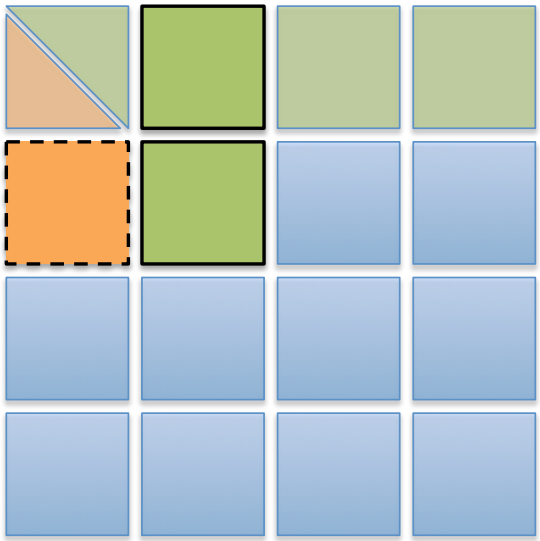
...

4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);



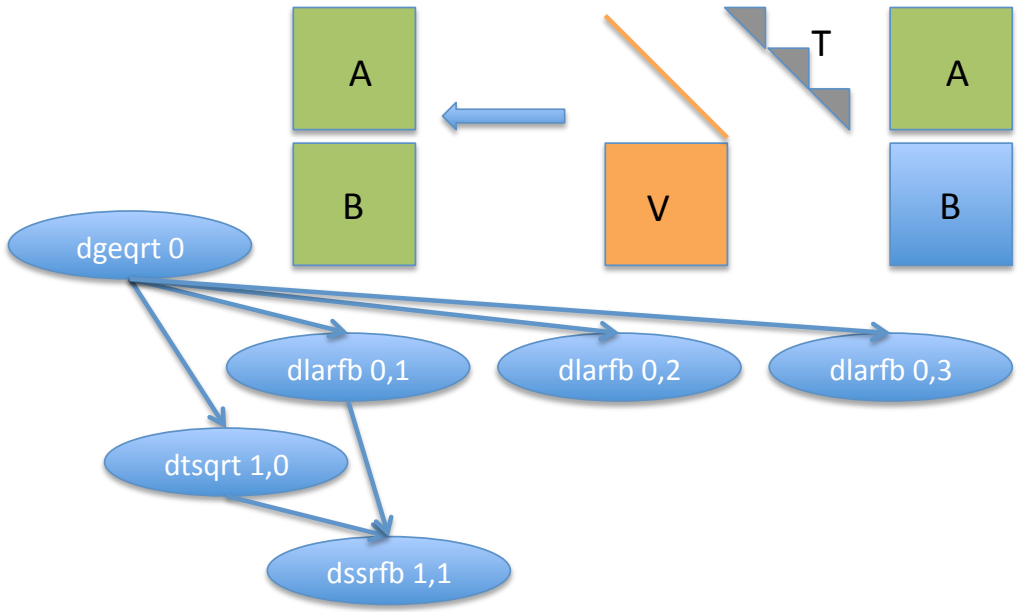

```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



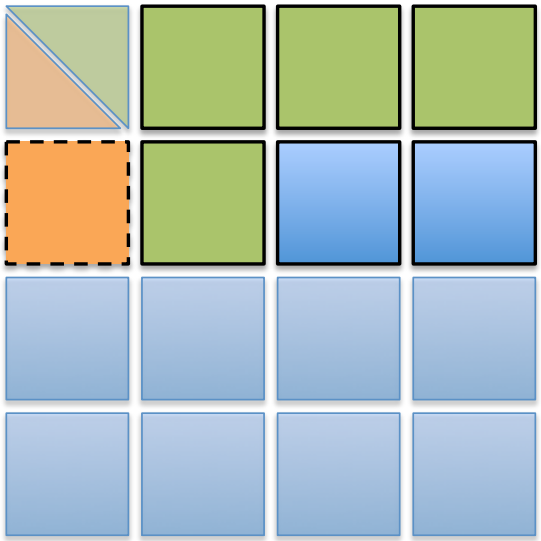
...

4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

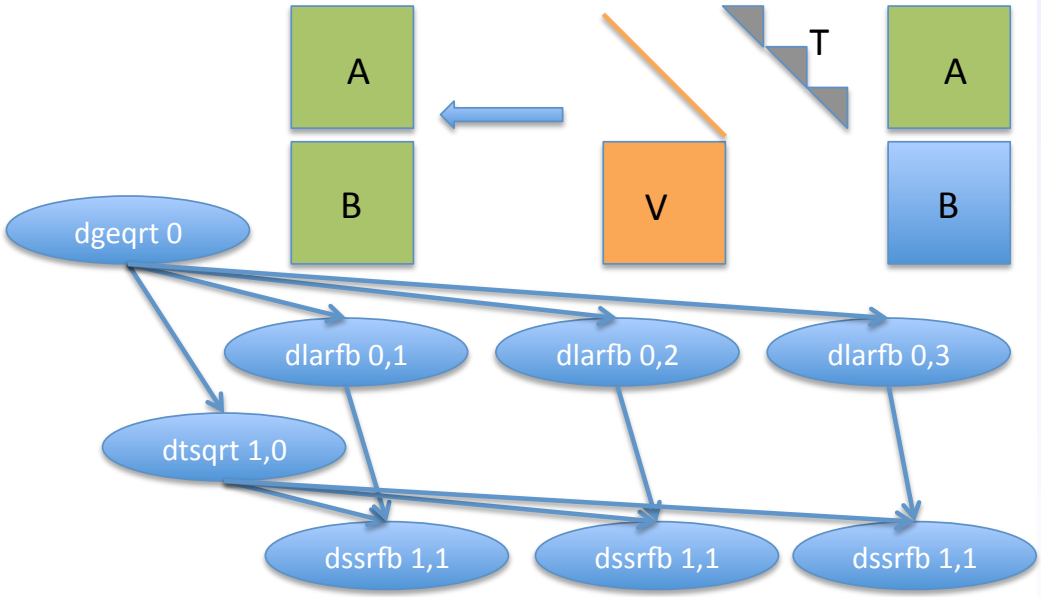



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree

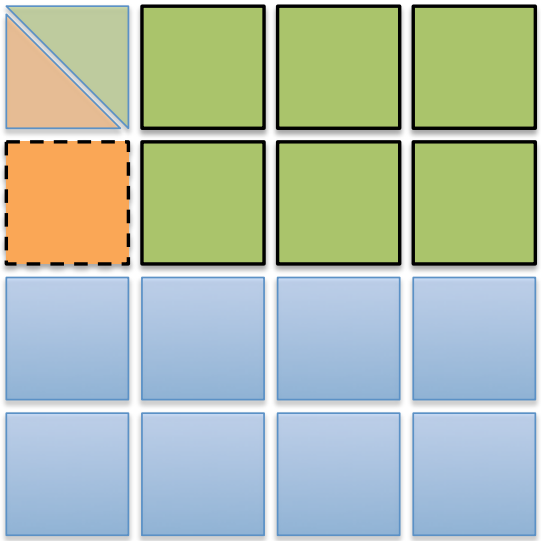


...
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
7. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
8. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

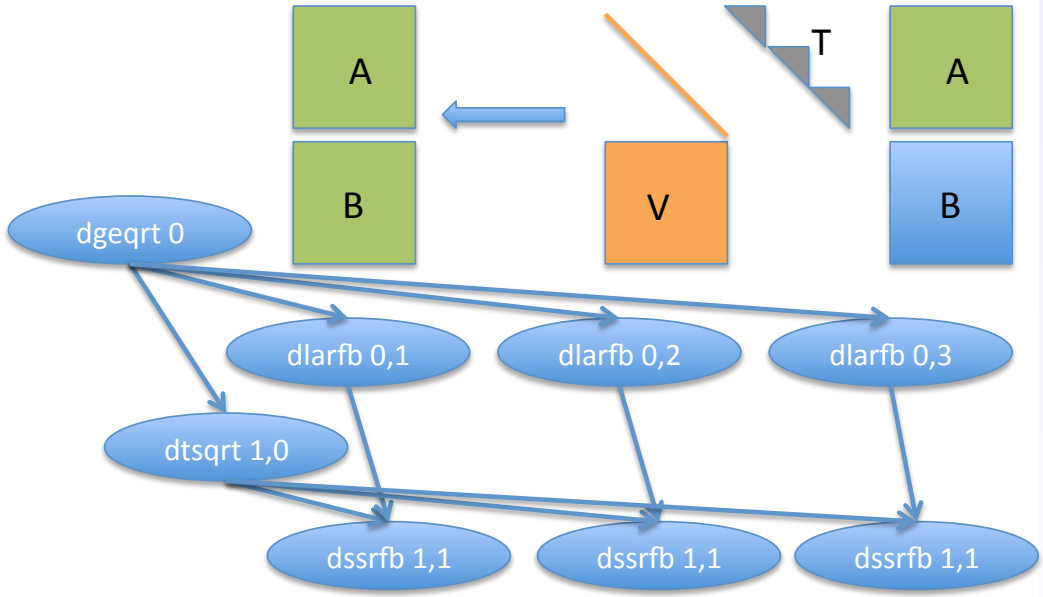



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree

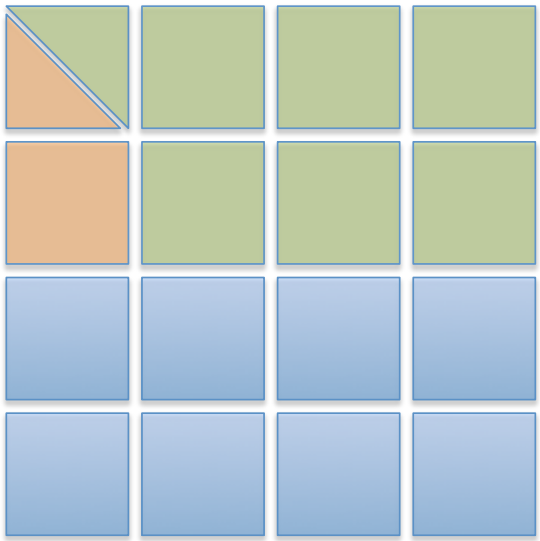


...
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
7. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
8. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

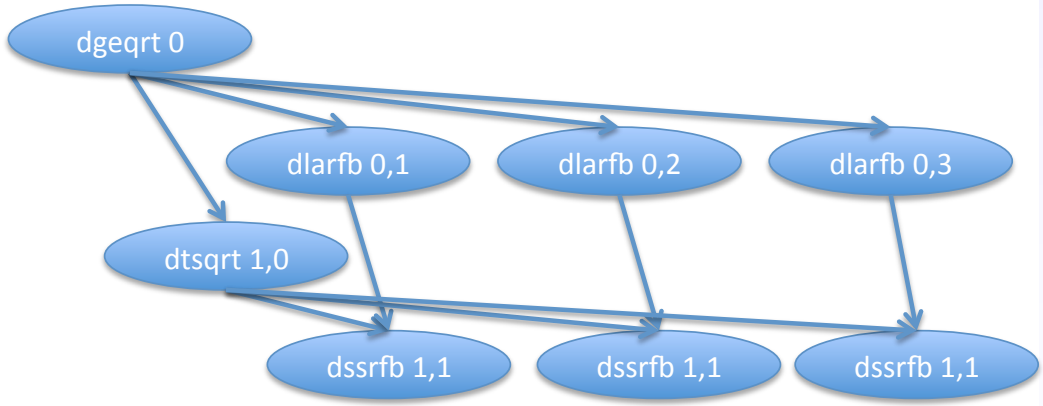



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# DAG - 2D tile QR - flat tree



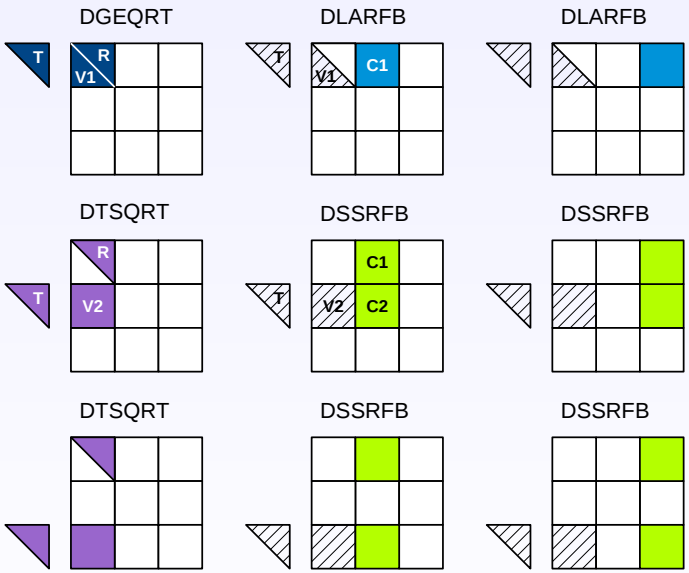
...
4. **dlarfb**(A[0][0], T[0][0], A[0][3]);
5. **dtsqrt**(A[0][0], A[1][0], T[1][0]);
6. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
7. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);
8. **dssrfb**(A[1][0], T[1][0], A[0][1], A[1][1]);

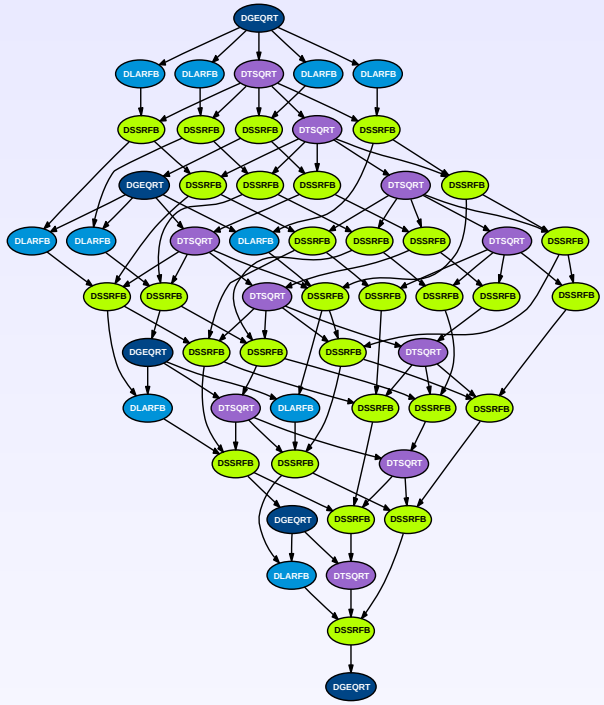



```
for (k = 0; k < TILES; k++) {
        dgeqrt(A[k][k], T[k][k]);
        for (n = k+1; n < TILES; n++) {
                dlarfb(A[k][k], T[k][k], A[k][n]);
        for (m = k+1; m < TILES; m++){
                dtsqrt(A[k][k], A[m][k], T[m][k]);
                for (n = k+1; n < TILES; n++)
                        dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
```

# 2D tile QR - flat tree - summary

```
FOR k = 0..TILES-1
   A[k][k], T[k][k] ← DGRQRT(A[k][k])
   FOR m = k+1..TILES-1
      A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
   FOR n = k+1..TILES-1
      A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
      FOR m = k+1..TILES-1
         A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

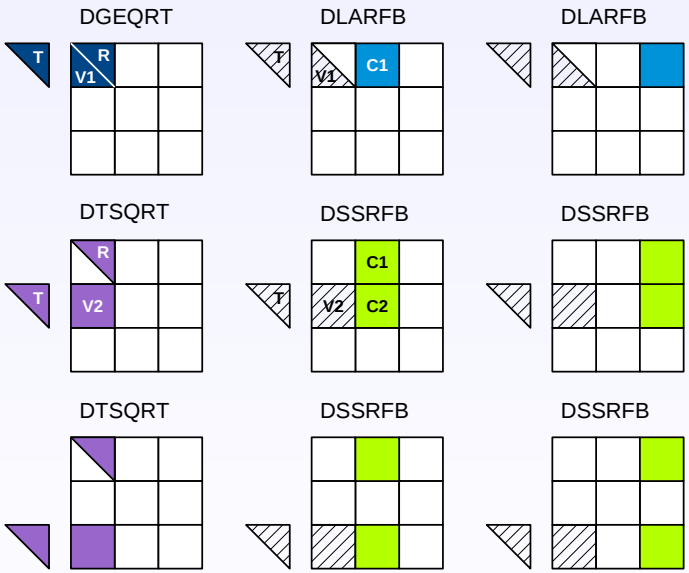


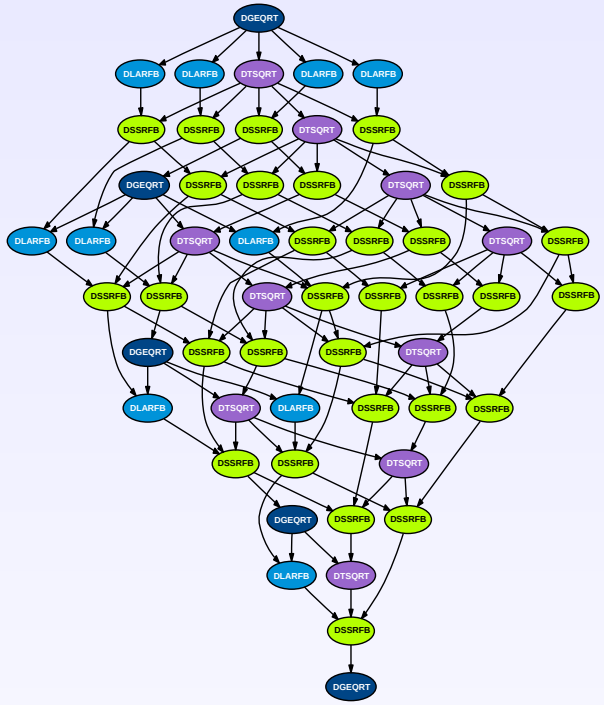




- Fine granularity;
- Tile layout;
- Different numerical properties;
- DAG to schedule.

# 2D tile QR - flat tree - summary

```
FOR k = 0..TILES-1
   A[k][k], T[k][k] ← DGRQRT(A[k][k])
   FOR m = k+1..TILES-1
      A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
   FOR n = k+1..TILES-1
      A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
      FOR m = k+1..TILES-1
         A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```




- Fine granularity;
- Tile layout;
- Different numerical properties;
- DAG to schedule.

# Outline

# Three-layers paradigm

## High-level algorithm

2D Tile QR - flat tree

## Runtime System

Intrusive (static) scheduler

## Device kernels

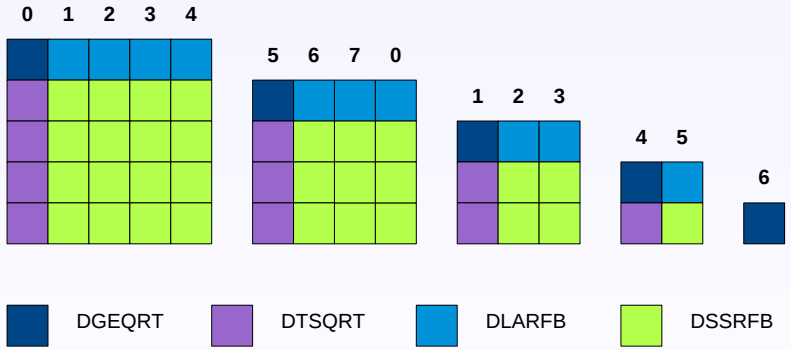
### CPU core

### GPU

# Implementation with a static pipeline (Plasma 2.0)

```
void dgeqrt(double *RV1, double *T);
void dtsqrt(double *R, double *V2, double *T);
void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);


k = 0; n = my_core_id;
while (n >= TILES) {
   k++; n = n-TILES+k;
} m = k;

while (k < TILES && n < TILES) {
   next_n = n; next_m = m; next_k = k;

   next_m++;
   if (next_m == TILES) {
      next_n += cores_num;
      while (next_n >= TILES && next_k < TILES) {
         next_k++; next_n = next_n-TILES+next_k;
      } next_m = next_k;
   }

   if (n == k) {
      if (m == k) {
         while(progress[k][k] != k-1);
         dgeqrt(A[k][k], T[k][k]);
         progress[k][k] = k;
      }
      else{
         while(progress[m][k] != k-1);
         dtsqrt(A[k][k], A[m][k], T[m][k]);
         progress[m][k] = k;
      }
   }
   else {
      if (m == k) {
         while(progress[k][k] != k);
         while(progress[k][n] != k-1);
         dlarfb(A[k][k], T[k][k], A[k][n]);
      }
      else {
         while(progress[m][k] != k);
         while(progress[m][n] != k-1);
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
         progress[m][n] = k;
      }
   }
   n = next_n; m = next_m; k = next_k;
}
```

- Work partitioned in one dimension (by block-columns).
- Cyclic assignment of work across all steps of the factorization (pipelining of factorization steps).
- Process tracking by a global progress table.
- Stall on dependencies (busy waiting).



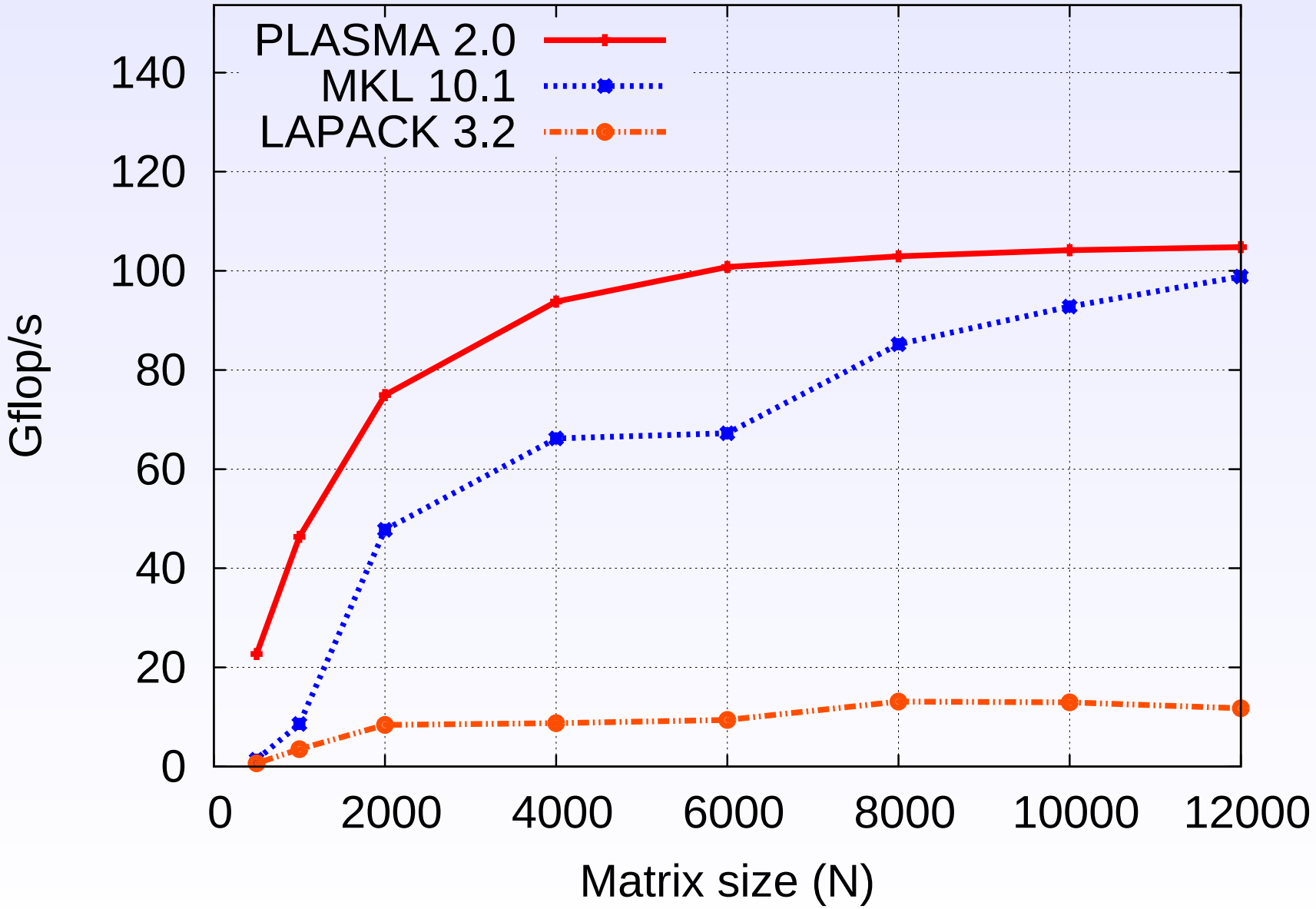
# Movies

- Plasma 1 core
- Plasma 4 cores

# Intel Xeon - 16 cores machine

- Node:
  - quad-socket quad-core Intel64 processors (16 cores).
- Intel Xeon processor:
  - quad-core;
  - frequency: 2,4 GHz.
- Theoretical peak:
  - 9.6 Gflop/s/core;
  - 153.6 Gflop/s/node.
- System and compilers:
  - Linux 2.6.25;
  - Intel Compilers 11.0.

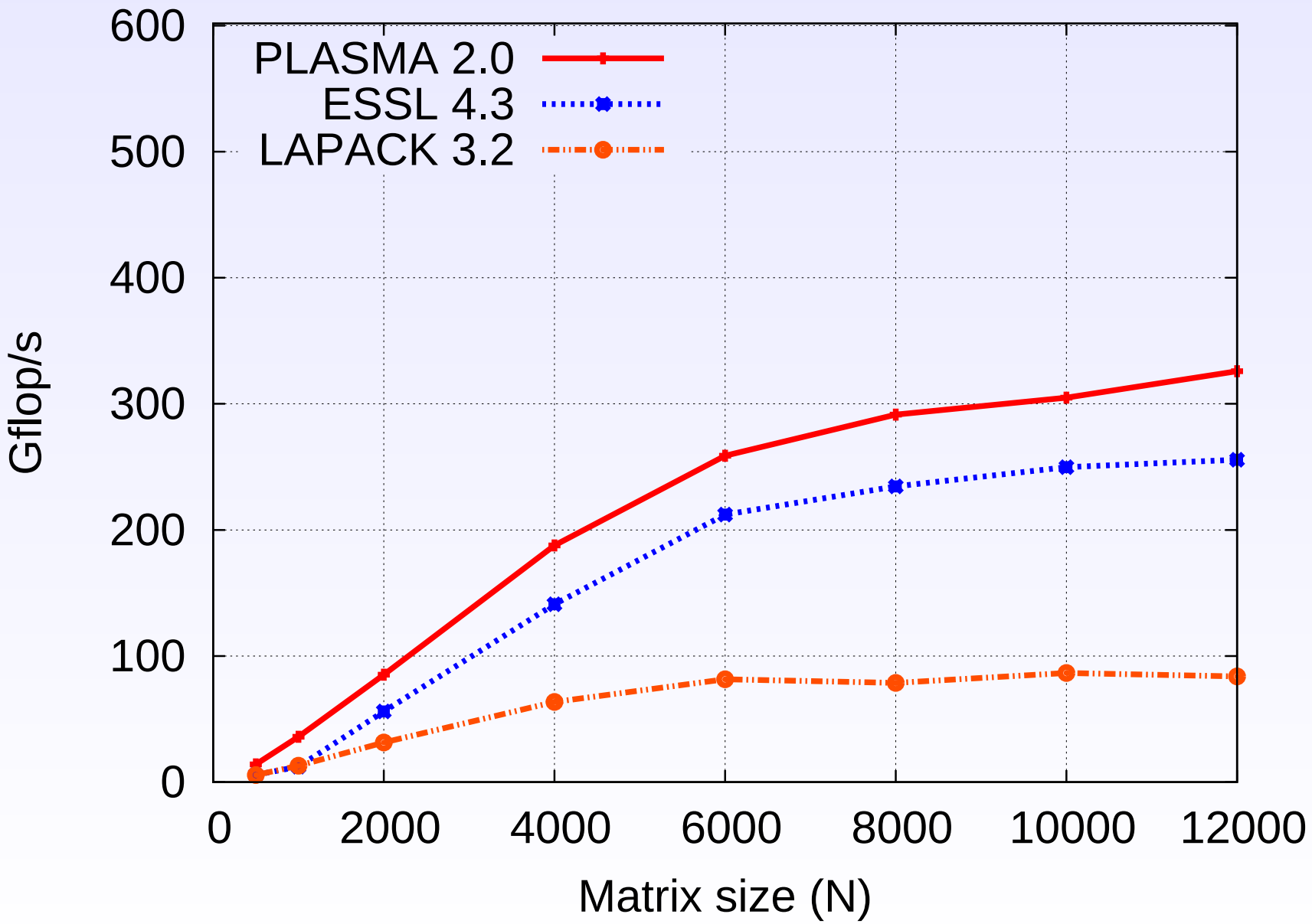
# Intel64- 16 cores - QR



# IBM Power6 - 32 cores machine

- Node:
  - 16 dual-core Power6 processors (32 cores).
- Power6 processor:
  - dual-core;
  - each core 2-way SMT;
  - L1: 64kB data + 64 kB instructions;
  - L2: 4 MB per core, accessible by the other core;
  - L3: 32 MB per processor, one controller per core (80 MB/s);
  - frequency: 4,7 GHz.
- Theoretical peak:
  - 18.8 Gflop/s/core;
  - 601.6 Gflop/s/node.
- System and compilers:
  - AIX 5.3;
  - xlf version 12.1;
  - xlc version 10.1.

# Power6- 32 cores - QR



# Three-layers paradigm

## High-level algorithm

## Runtime System

DAG vs Fork-Join

## Device kernels

### CPU core

### GPU

# A few runtime systems

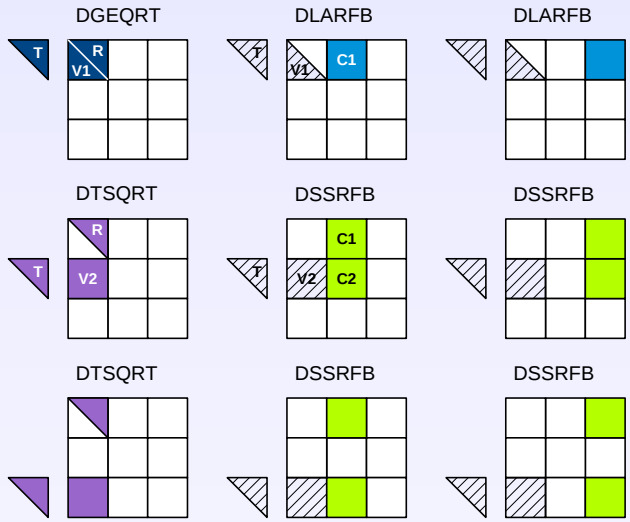
- Cilk/Cilk++ [Fork-join];
- SMP Superscalar (SMPSs) [DAG] – GPUSs – StarSs;
- StarPU;
- Quark (Plasma, since version 2.1);
- DAGuE (dPlasma);
- SuperMatrix;
- Intel Threading Building Blocks;
- Charm++;
- ...

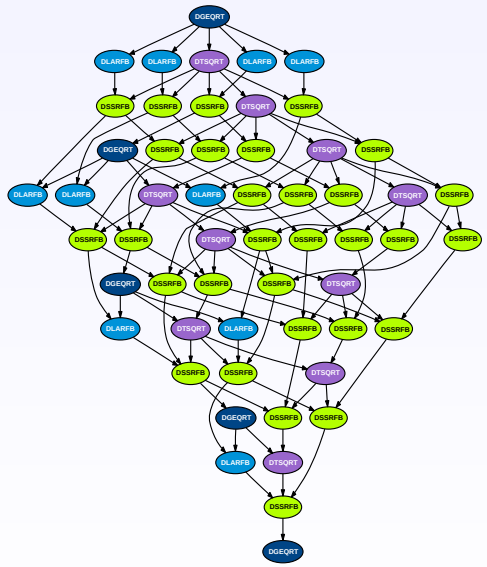
# A few runtime systems

- Cilk/Cilk++ [Fork-join];
- SMP Superscalar (SMPSs) [DAG] – GPUSs – StarSs;
- StarPU;
- Quark (Plasma, since version 2.1);
- DAGuE (dPlasma);
- SuperMatrix;
- Intel Threading Building Blocks;
- Charm++;
- ...

# Fork-Join model - Cilk - 2D

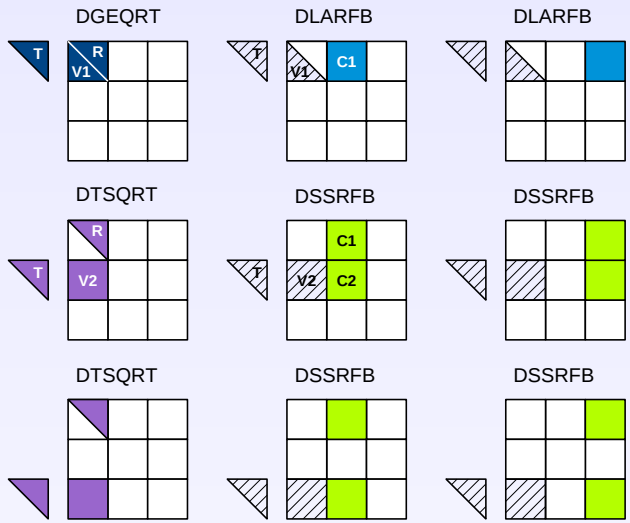
```
cilk void dgeqrt(double *RV1, double *T);
cilk void dtsqrt(double *R, double *V2, double *T);
cilk void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);

cilk void dssrfb_(int m, int n, int k)
{
   dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);

   if (m == TILES-1 && n == k+1 && k+1 < TILES)
      spawn dgeqrt(A[k+1][k+1], T[k+1][k+1]);

   if (n == k+1 && m+1 < TILES)
      spawn dtsqrt(A[k][k], A[m+1][k], T[m+1][k]);
}


spawn dgeqrt(A[0][0], T[0][0]);
sync;

for (k = 0; k < TILES; k++) {

   for (n = k+1; n < TILES; n++)
      spawn dlarfb(A[k][k], T[k][k], A[k][n]);

   if (k+1 < TILES)
      spawn dtsqrt(A[k][k], A[k+1][k], T[k+1][k]);
   sync;

   for (m = k+1; m < TILES; m++) {
      for (n = k+1; n < TILES; n++)
         spawn dssrfb_(m, n, k);
      sync;
   }
}
```

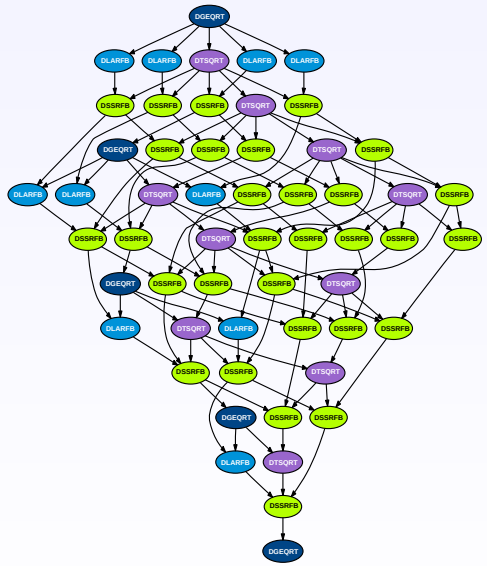


# Fork-Join model - Cilk - 1D

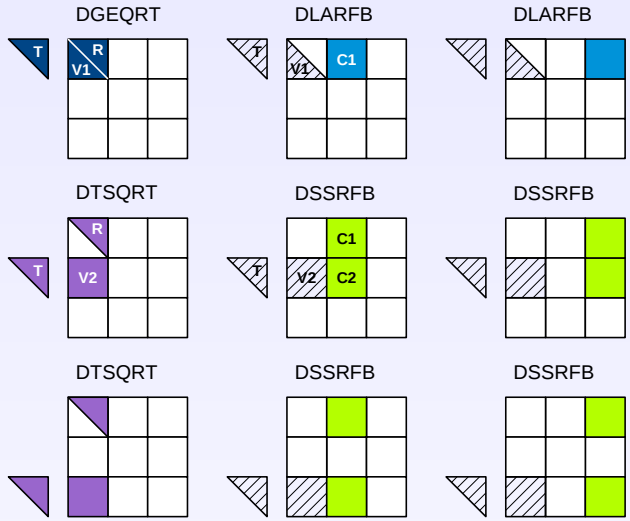
```
void dgeqrt(double *RV1, double *T);
void dtsqrt(double *R, double *V2, double *T);
void dlarfb(double *V1, double *T, double *C1);
void dssrfb(double *V2, double *T, double *C1, double *C2);

cilk void qr_panel(int k)
{
   int m;

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k], A[m][k], T[m][k]);
}

cilk void qr_update(int n, int k)
{
   int m;

   dlarfb(A[k][k], T[k][k], A[k][n]);

   for (m = k+1; m < TILES; m++)
      dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);

   if (n == k+1)
      spawn qr_panel(k+1);
}


spawn qr_panel(0);
sync;

for (k = 0; k < TILES; k++) {
   for (n = k+1; n < TILES; n++)
      spawn qr_update(n, k);
   sync;
}
```

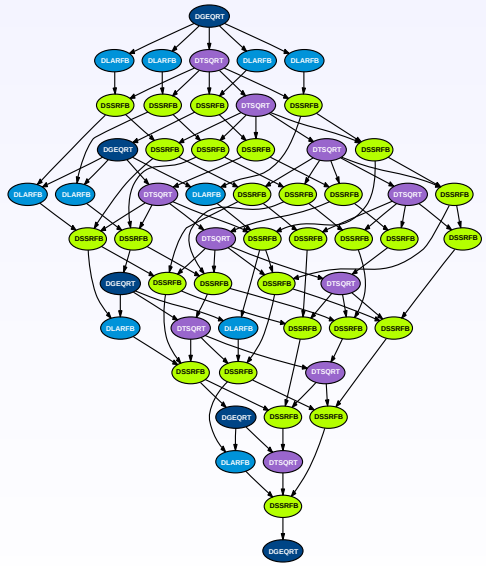


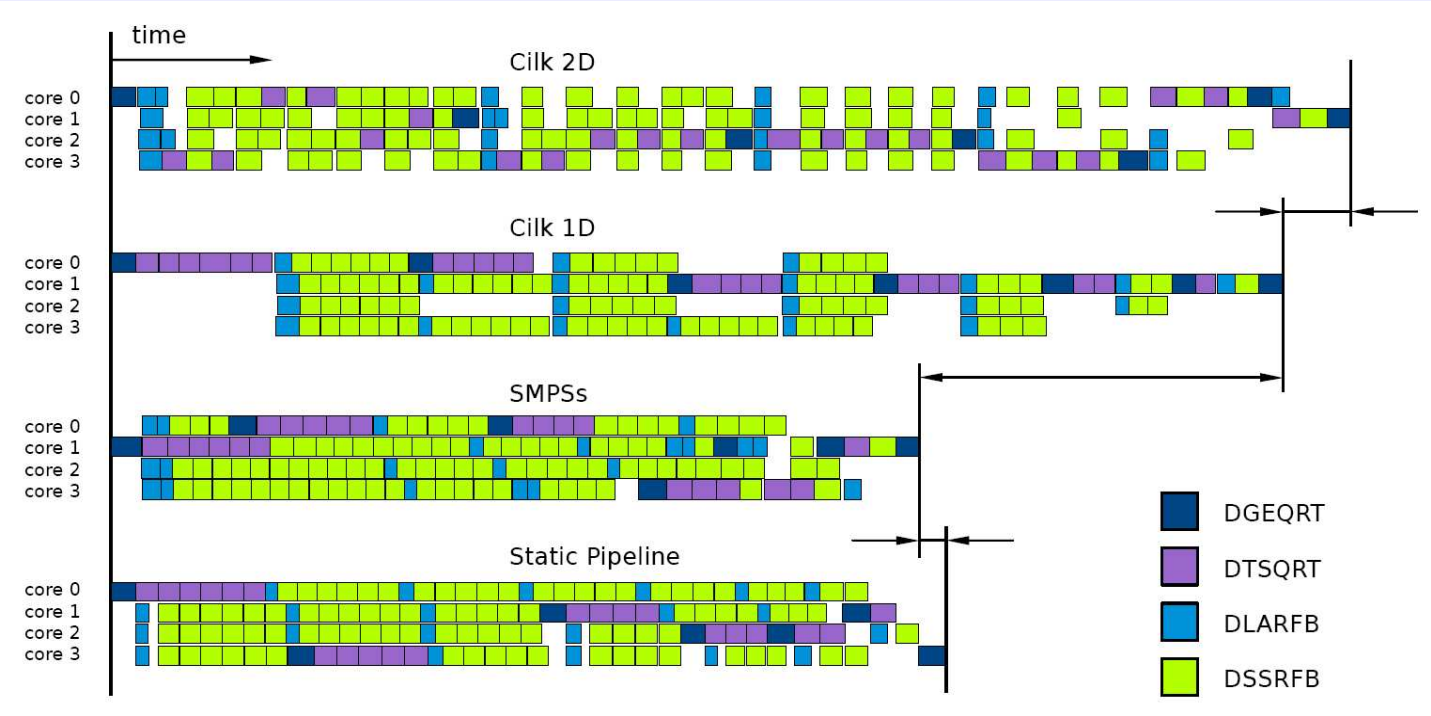
# DAG - SMPSs

```
#pragma css task \
   inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task \
   inout(R[◥], V2[NB][NB]) output(T[NB][NB])
void dtsqrt(double *R, double *V2, double *T);

#pragma css task \
   input(V1[◣], T[NB][NB]) inout(C1[NB][NB])
void dlarfb(double *V1, double *T, double *C1);

#pragma css task \
   input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void dssrfb(double *V2, double *T, double *C1, double *C2);


#pragma css start
for (k = 0; k < TILES; k++) {

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k]◥, A[m][k], T[m][k]);

   for (n = k+1; n < TILES; n++) {
      dlarfb(A[k][k]◣, T[k][k], A[k][n]);
      for (m = k+1; m < TILES; m++)
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
   }
}
#pragma css finish
```

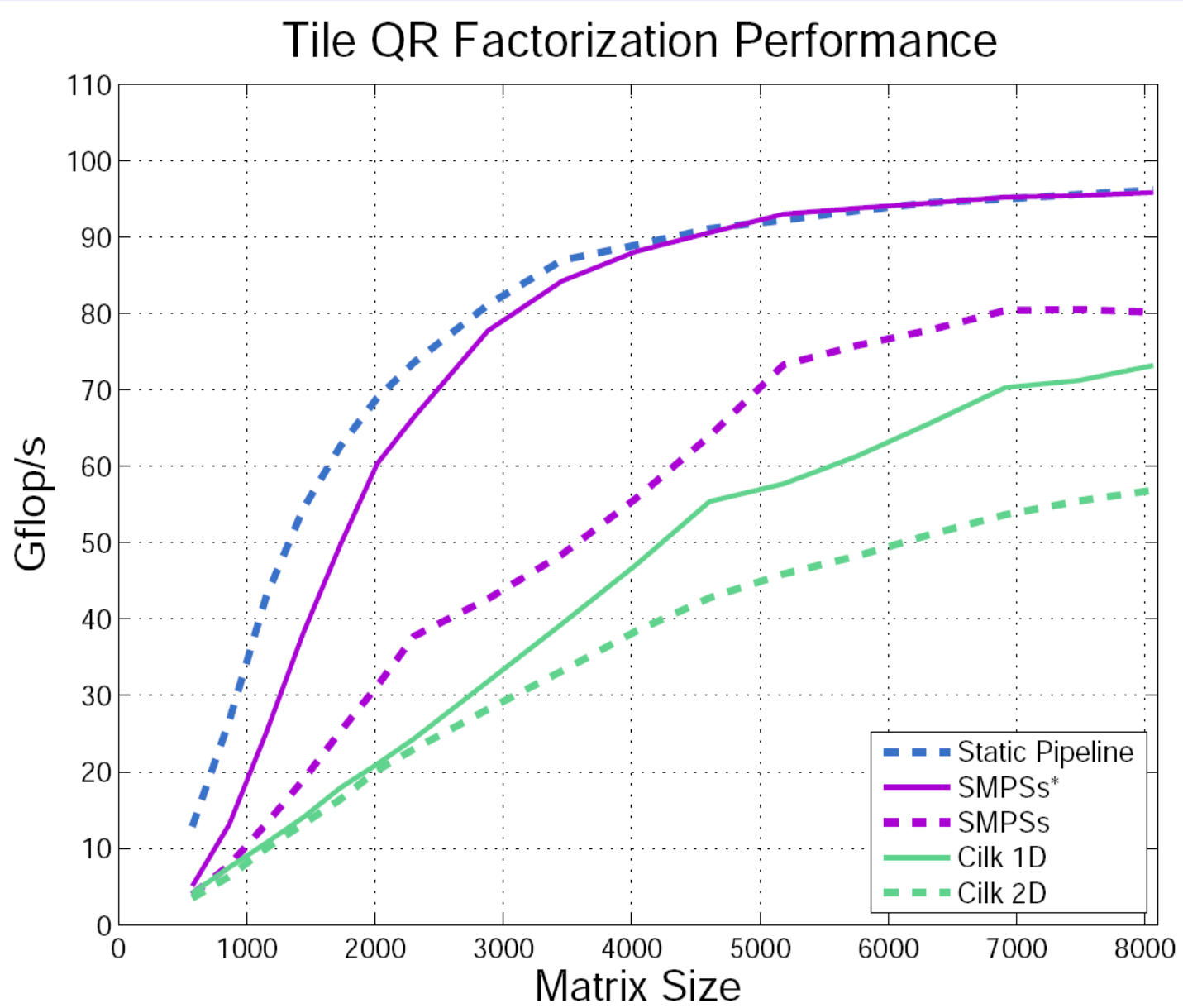


# Traces [Kurzak et al.'09]



# Performance - Intel 16 cores [Kurzak et al.'09]



# Outline

# Three-layers paradigm

High-level algorithm

Runtime System

Device kernels

CPU core

GPU

# Three-layers paradigm

High-level algorithm

Runtime System

Device kernels

CPU core

GPU

# Three-layers paradigm

High-level algorithm

Runtime System

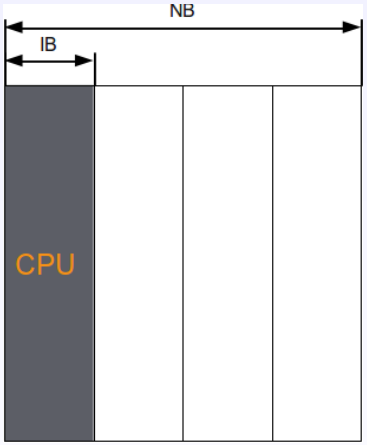
Device kernels

CPU core

GPU

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



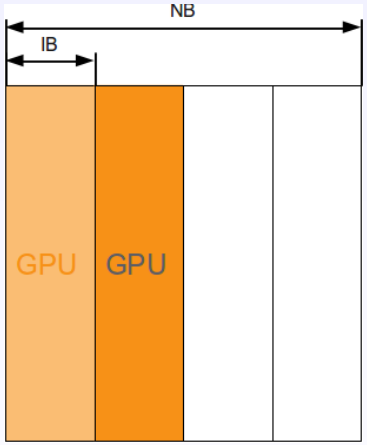

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



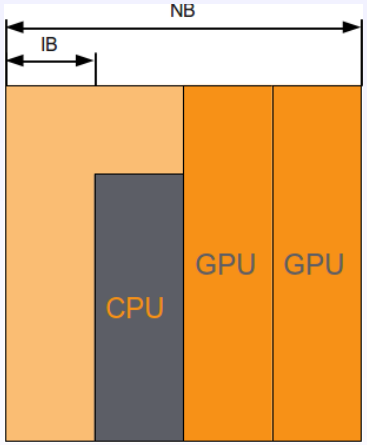

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



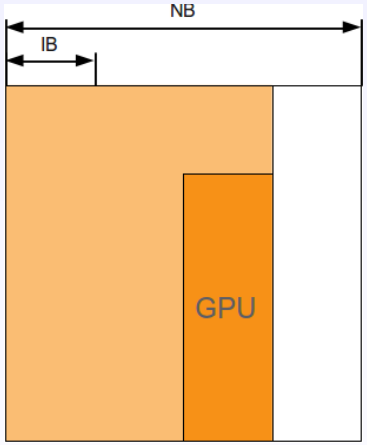

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



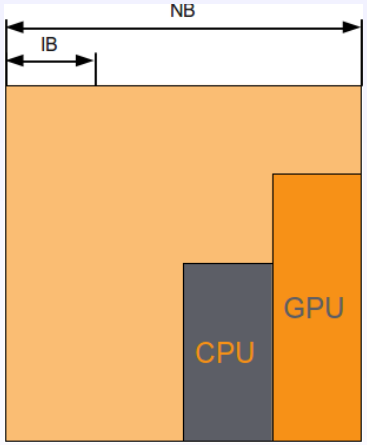

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



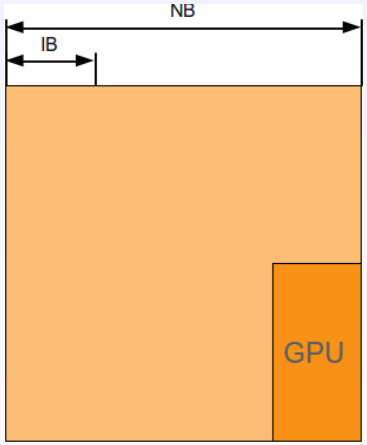

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU



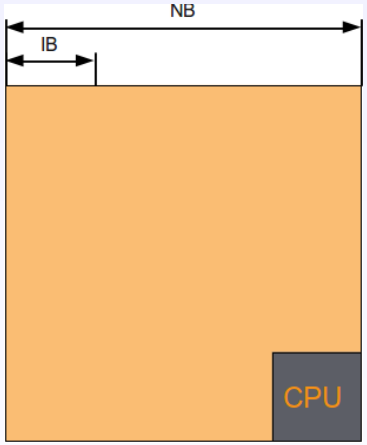

CUDA `tsmqr` kernel:

$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

# GPU kernels

- Kernels for update (`ormqr` and `tsmqr`)
  Fully on GPU
- Kernels for panel factorization (`geqrt` and `tsqrt`)
  Hybrid implementation CPU + GPU




CUDA `tsmqr` kernel:

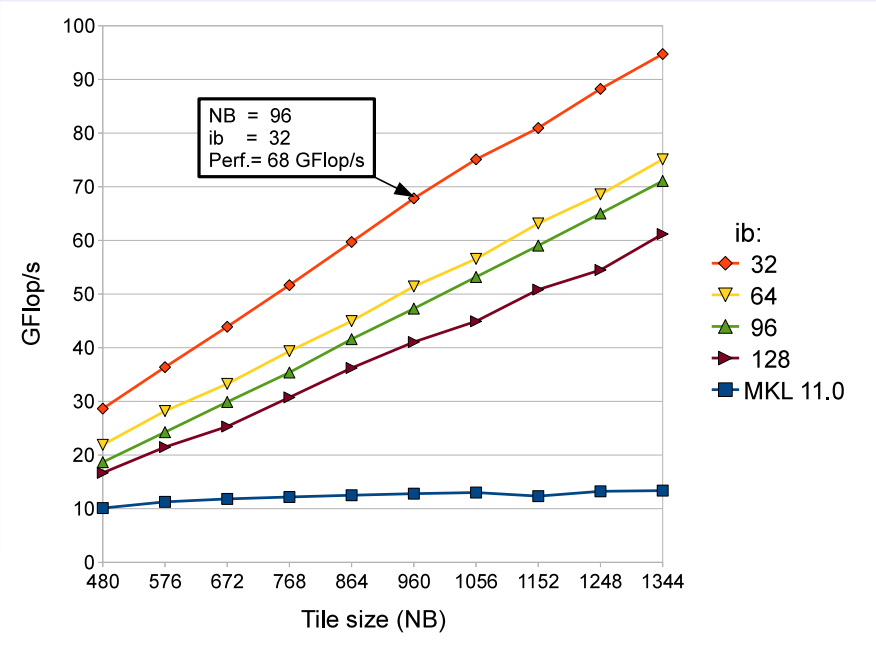
$$\begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix} = \left[ I - \begin{pmatrix} I \\ V_j \end{pmatrix} T_j \begin{pmatrix} I \\ V_j \end{pmatrix}^T \right] \begin{pmatrix} A^j_{ki} \\ A_{mi} \end{pmatrix},$$

1. $D^1_{work} = A^j_{ki} + V_j^T A_{mi}$;
2. $D^2_{work} = T_j D^1_{work}$; $A^j_{ki} = A^j_{ki} - D^2_{work}$;
3. $A_{mi} = A_{mi} - V_j D^2_{work}$.

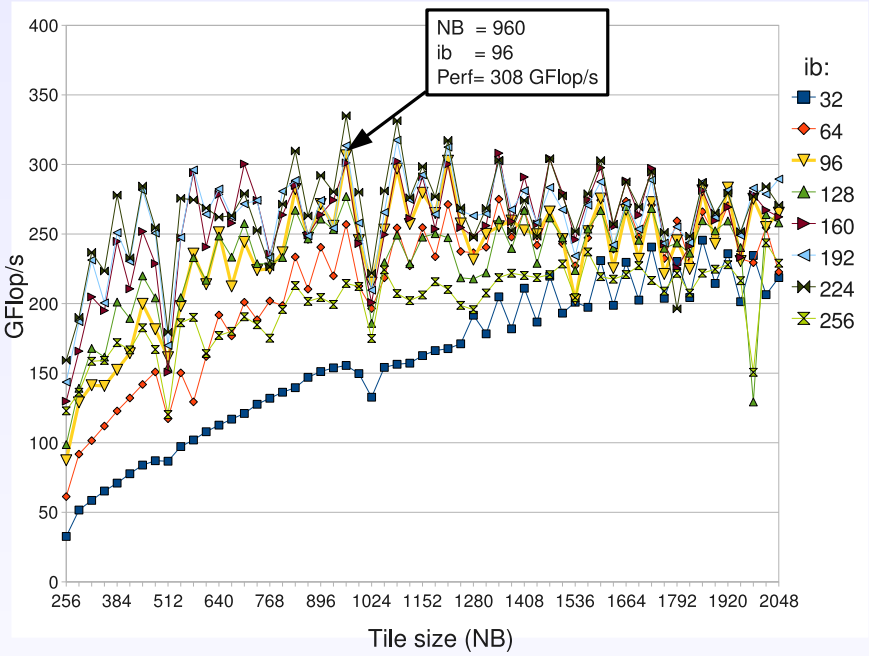
# Architecture

- AMD Opteron 8358 SE CPU, 4 $\times$ 4, 2.4GHz, 4 $\times$ 8GB
- NVIDIA Tesla S1070 GPU, 4 $\times$ 240, 1.3GHz, 4 $\times$ 4GB
- single precision:
  - peak: 3067Gflop/s (307.2 + 2760)
  - `sgemm`: 1908Gflop/s (256 + 1652)
- double precision:
  - peak: 498.6Gflop/s (153.6 + 345)
  - `dgemm`: 467.2Gflop/s (131.2 + 336)

# Tuning

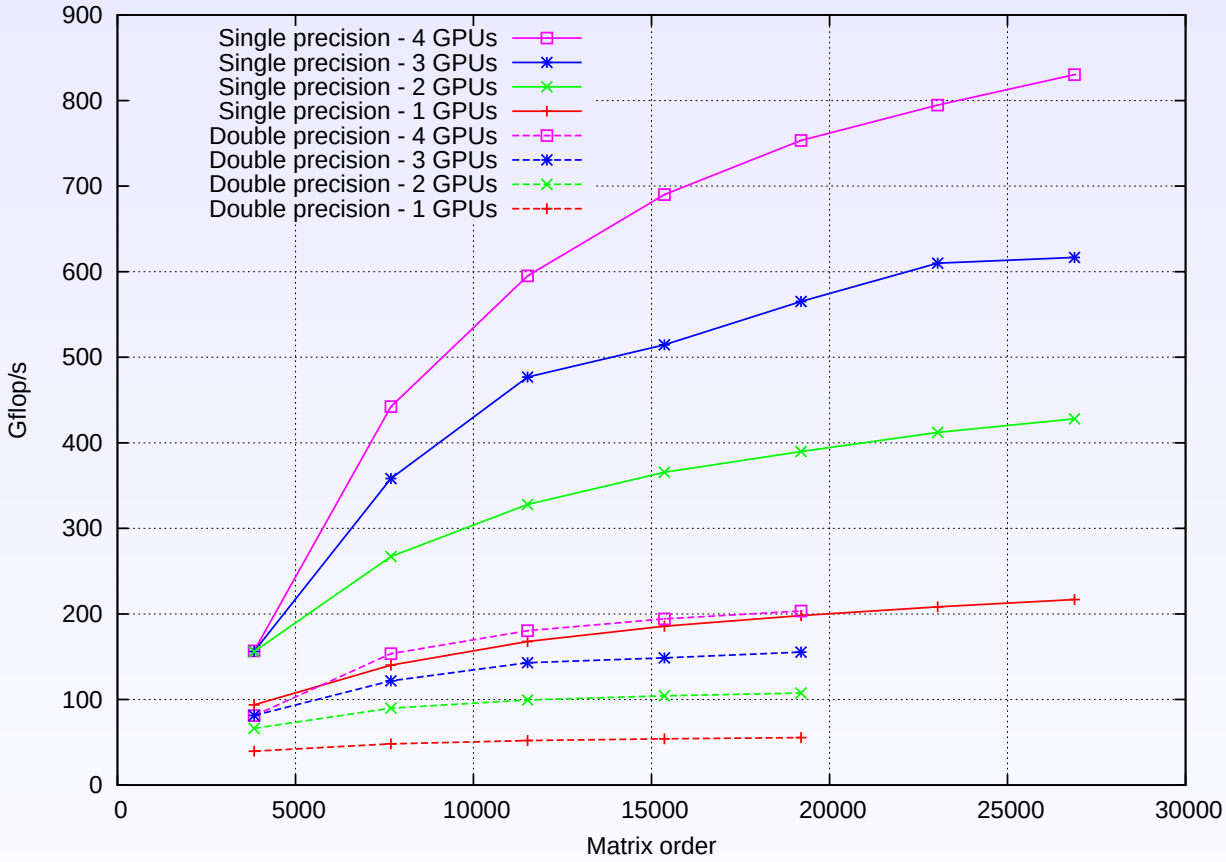



Panel factorization
(`sgeqrt` kernel)



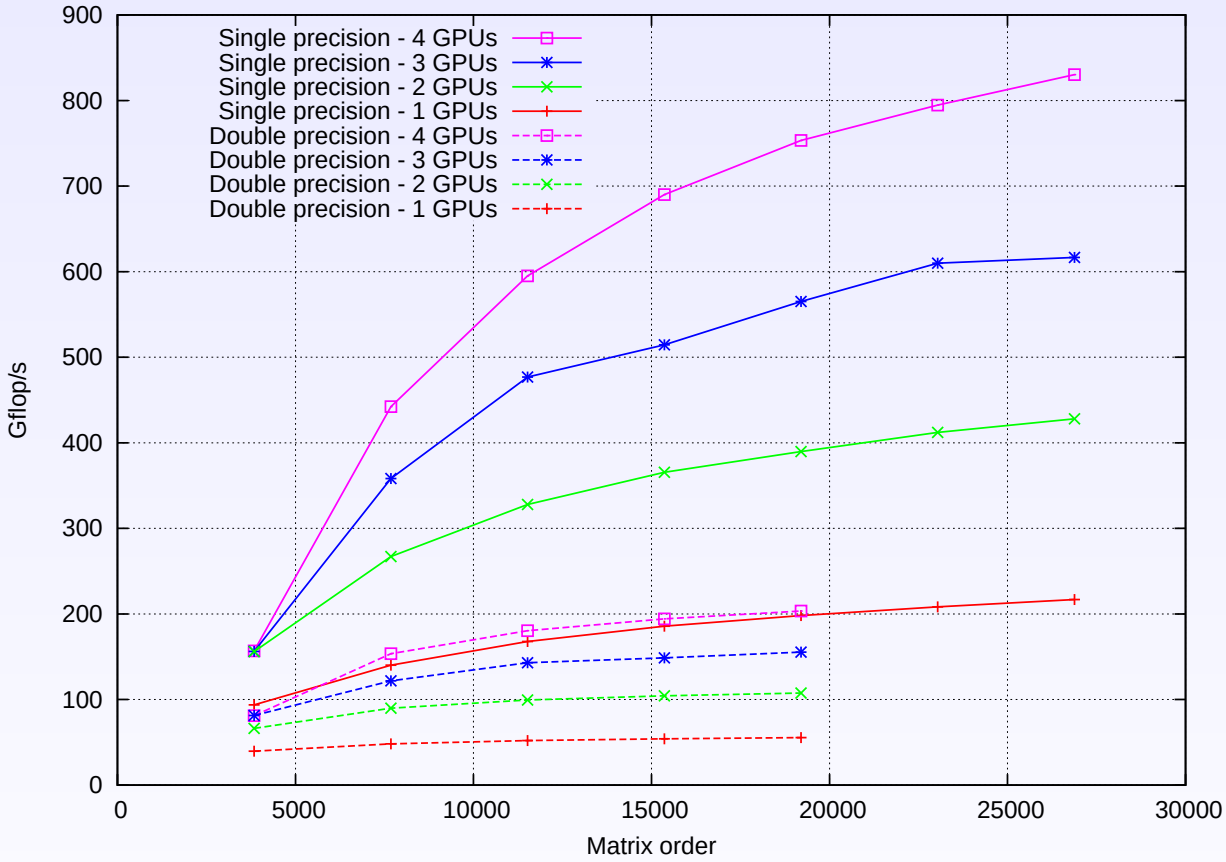

Update phase
(`stsmqr` kernel)

# Performance




How to use the 16 CPU cores?

# Performance




How to use the 16 CPU cores?

# Three-layers paradigm

## High-level algorithm

## Runtime System

StarPU

## Device kernels

### CPU core

### GPU

# GPU-enabled runtime systems

- Cilk/Cilk++;
- SMP Superscalar (SMPSs) – GPUSs – StarSs;
- StarPU;
- Quark (Plasma, since version 2.1);
- DAGuE (dPlasma);
- SuperMatrix;
- Intel Threading Building Blocks;
- Charm++;
- ...

# GPU-enabled runtime systems

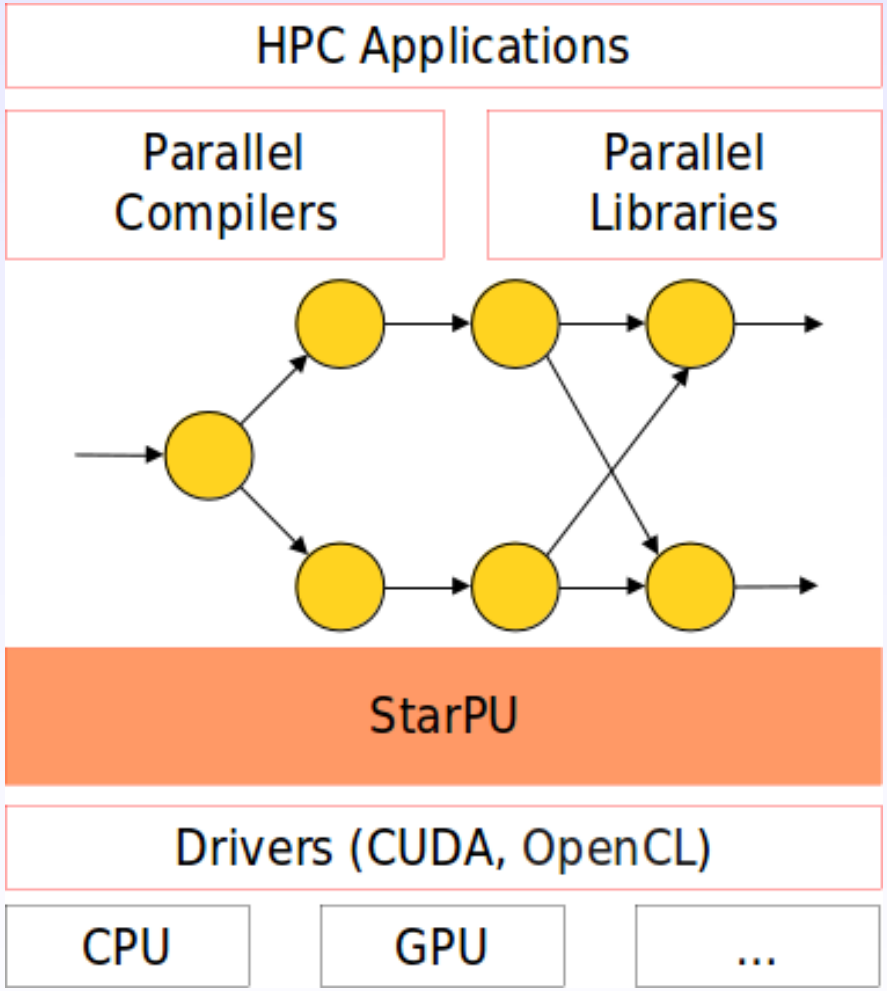
- Cilk/Cilk++;
- SMP Superscalar (SMPSs) – GPUSs – StarSs;
- StarPU;
- Quark (Plasma, since version 2.1);
- DAGuE (dPlasma);
- SuperMatrix;
- Intel Threading Building Blocks;
- Charm++;
- ...

# GPU-enabled runtime systems

- Cilk/Cilk++;
- SMP Superscalar (SMPSs) – GPUSs – StarSs;
- StarPU;
- Quark (Plasma, since version 2.1);
- DAGuE (dPlasma);
- SuperMatrix;
- Intel Threading Building Blocks;
- Charm++;
- ...

# The StarPU runtime system [Augonnet et al.]

- Data management:
  - Checks dependences;
  - Ensures coherency;
- Supports:
  - SMP/Multicore Processors (x86, PPC, ...);
  - NVIDIA GPUs;
  - OpenCL devices;
  - Cell Processors (experimental).
- Scheduling module.



# 2D Tile QR - flat tree - over StarPU

```
for (k = 0; k < min(MT, NT); k++){
  starpu_Insert_Task(&cl_zgeqrt, k , k, ...);

  for (n = k+1; n < NT; n++)
    starpu_Insert_Task(&cl_zunmqr, k, n, ...);

  for (m = k+1; m < MT; m++){
    starpu_Insert_Task(&cl_ztsqrt, m, k, ...);

    for (n = k+1; n < NT; n++)
      starpu_Insert_Task(&cl_ztsmqr, m, n, k, ...);
  }
}
```

See code

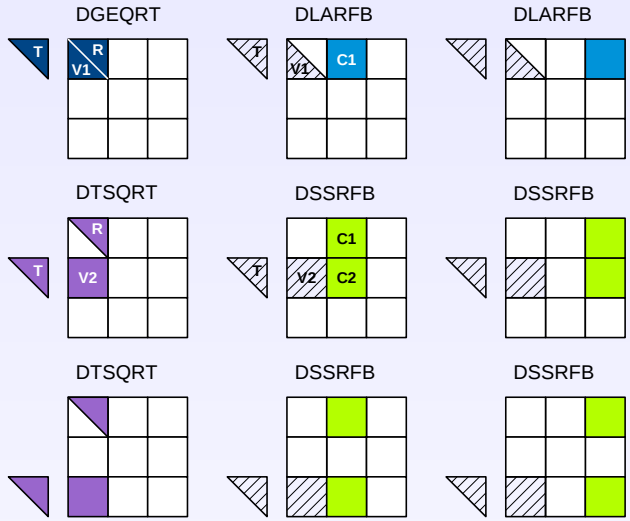
# 2D Tile QR - flat tree - over StarPU

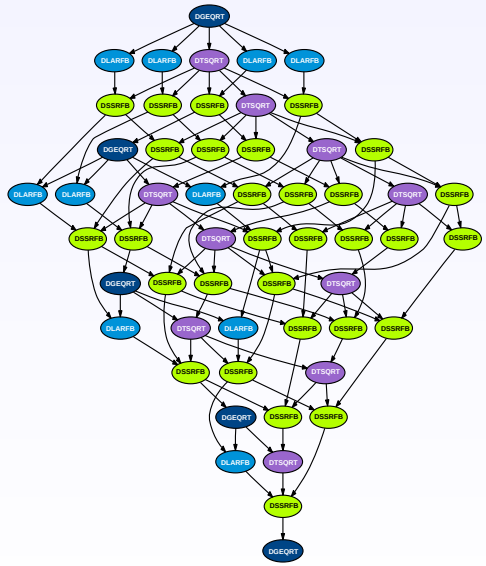
```
for (k = 0; k < min(MT, NT); k++){
  starpu_Insert_Task(&cl_zgeqrt, k , k, ...);

  for (n = k+1; n < NT; n++)
    starpu_Insert_Task(&cl_zunmqr, k, n, ...);

  for (m = k+1; m < MT; m++){
    starpu_Insert_Task(&cl_ztsqrt, m, k, ...);

    for (n = k+1; n < NT; n++)
      starpu_Insert_Task(&cl_ztsmqr, m, n, k, ...);
  }
}
```

See code

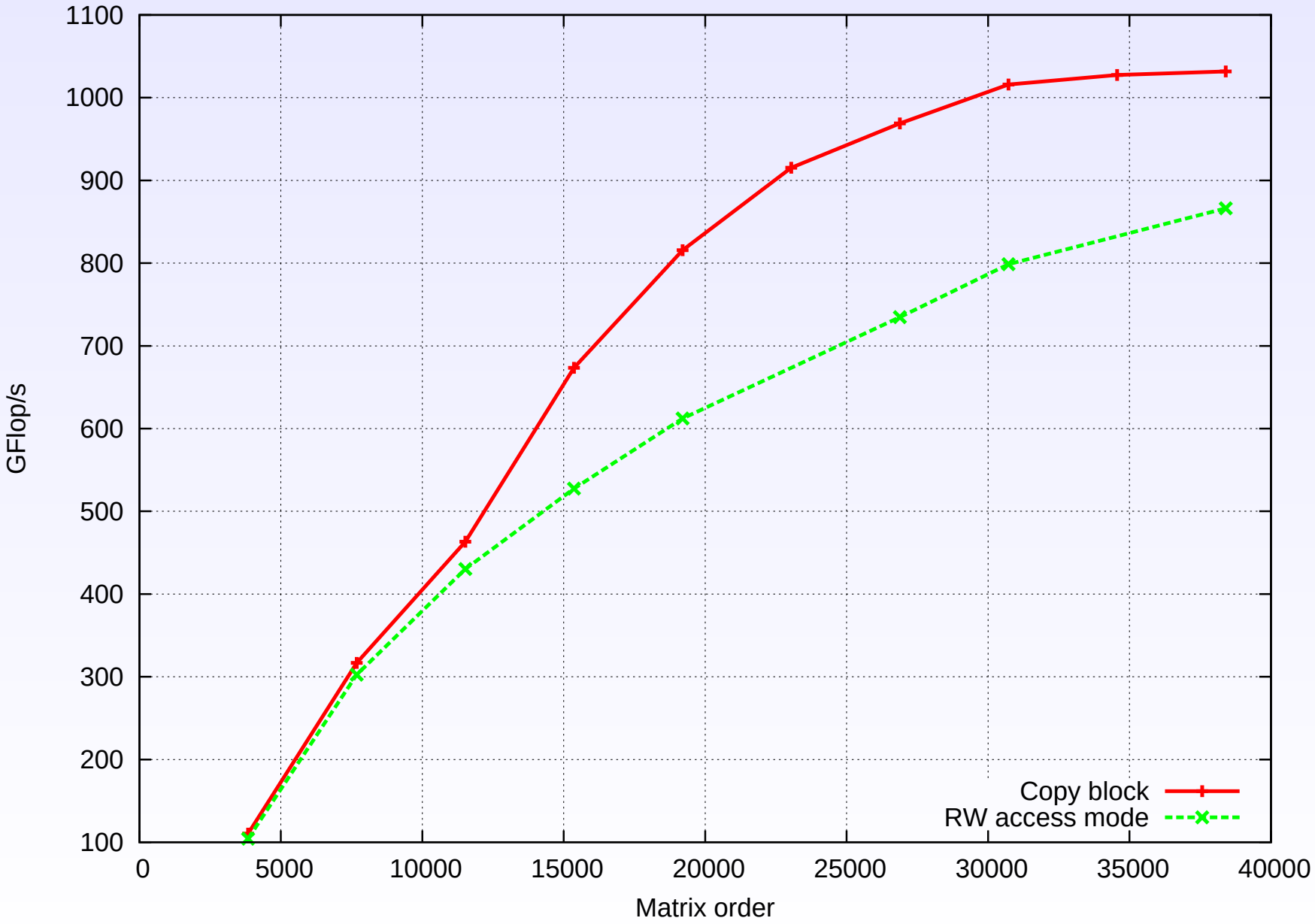
# Relieving anti-dependencies (SMPSs trick reminder)

```
#pragma css task \
   inout(RV1[NB][NB]) output(T[NB][NB])
void dgeqrt(double *RV1, double *T);

#pragma css task \
   inout(R[◥], V2[NB][NB]) output(T[NB][NB])
void dtsqrt(double *R, double *V2, double *T);

#pragma css task \
   input(V1[◣], T[NB][NB]) inout(C1[NB][NB])
void dlarfb(double *V1, double *T, double *C1);

#pragma css task \
   input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void dssrfb(double *V2, double *T, double *C1, double *C2);


#pragma css start
for (k = 0; k < TILES; k++) {

   dgeqrt(A[k][k], T[k][k]);

   for (m = k+1; m < TILES; m++)
      dtsqrt(A[k][k]◥, A[m][k], T[m][k]);

   for (n = k+1; n < TILES; n++) {
      dlarfb(A[k][k]◣, T[k][k], A[k][n]);
      for (m = k+1; m < TILES; m++)
         dssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
   }
}
#pragma css finish
```

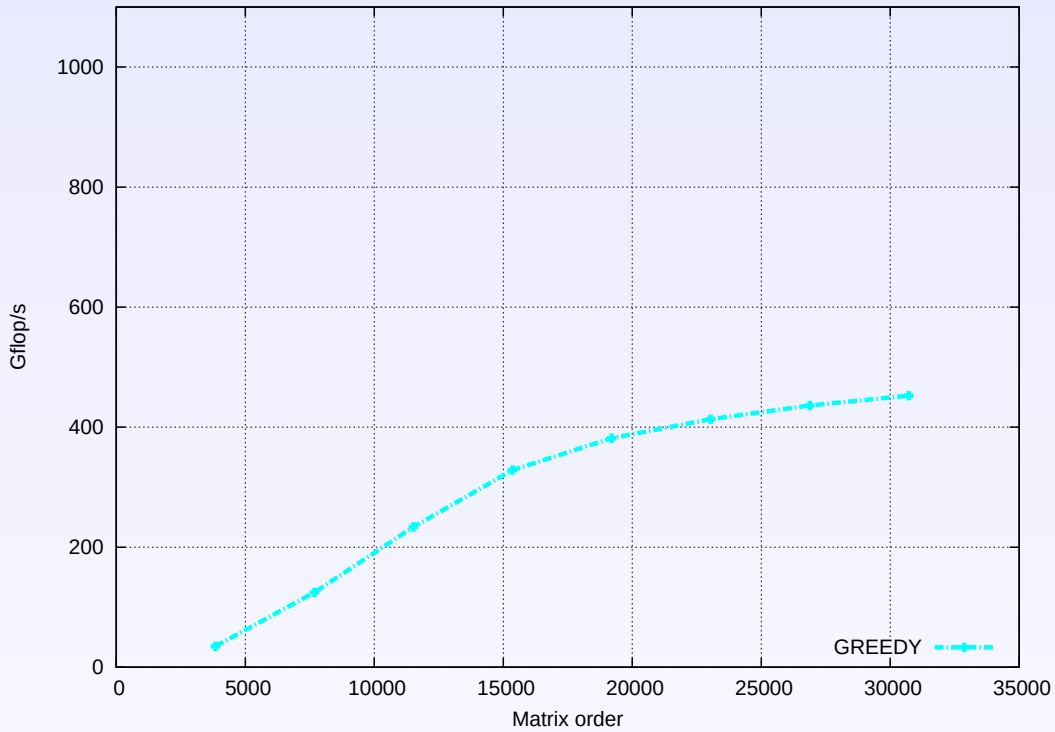


# Relieving anti-dependencies (using StarPU)

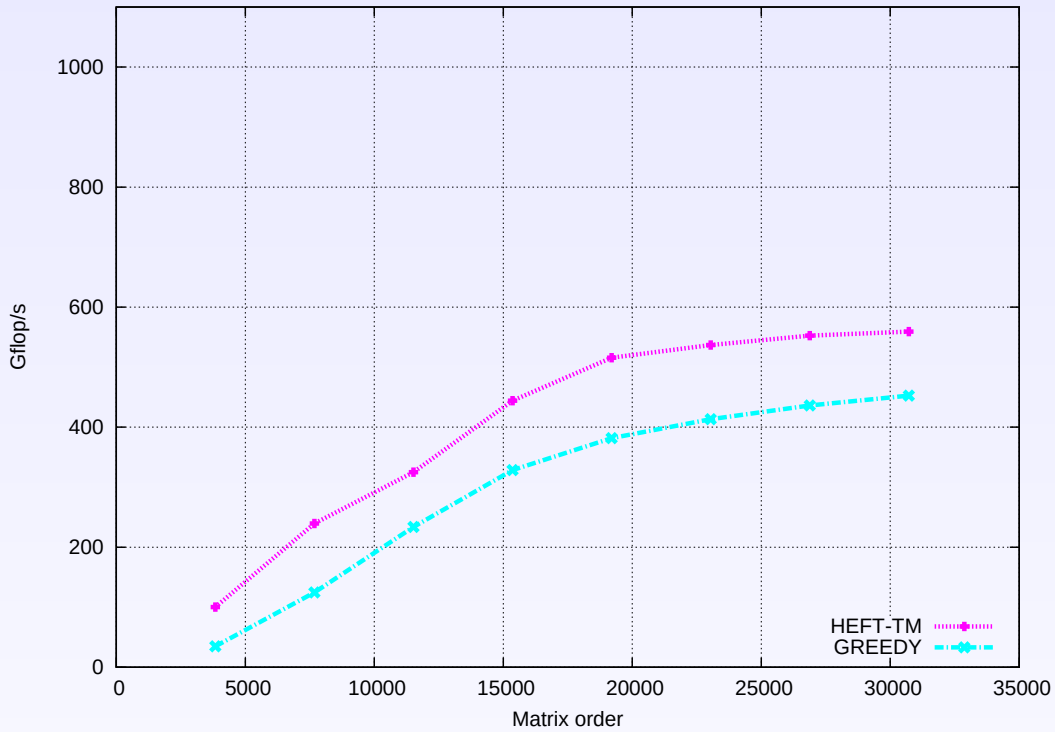


# Impact of the scheduling policy



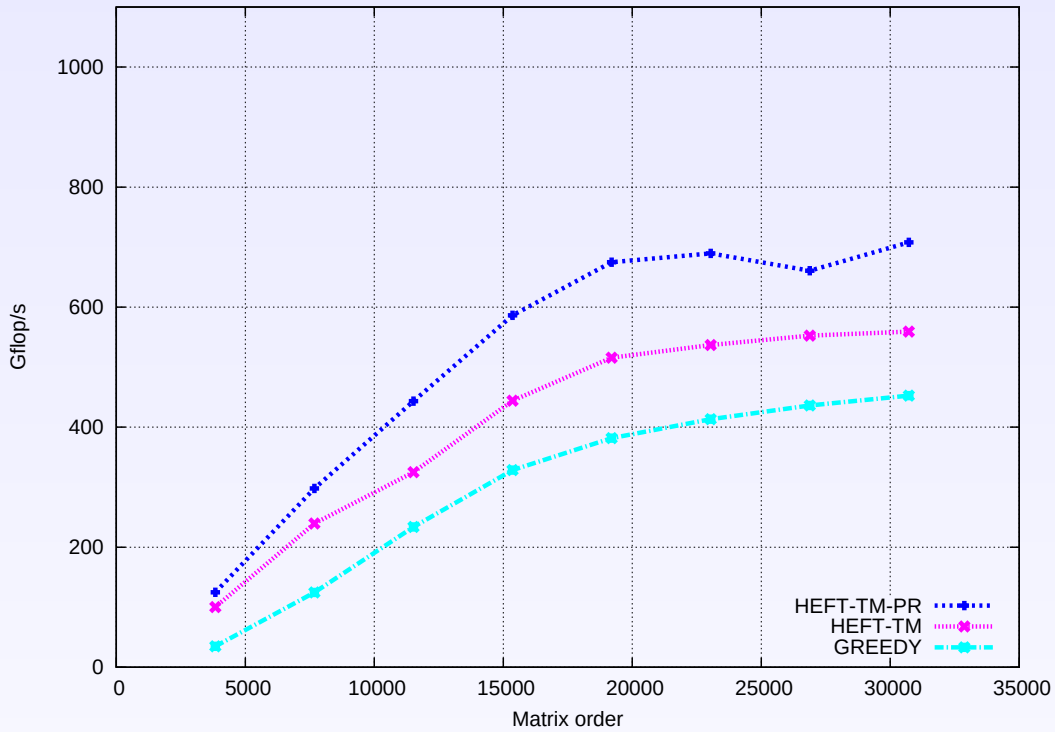

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the scheduling policy



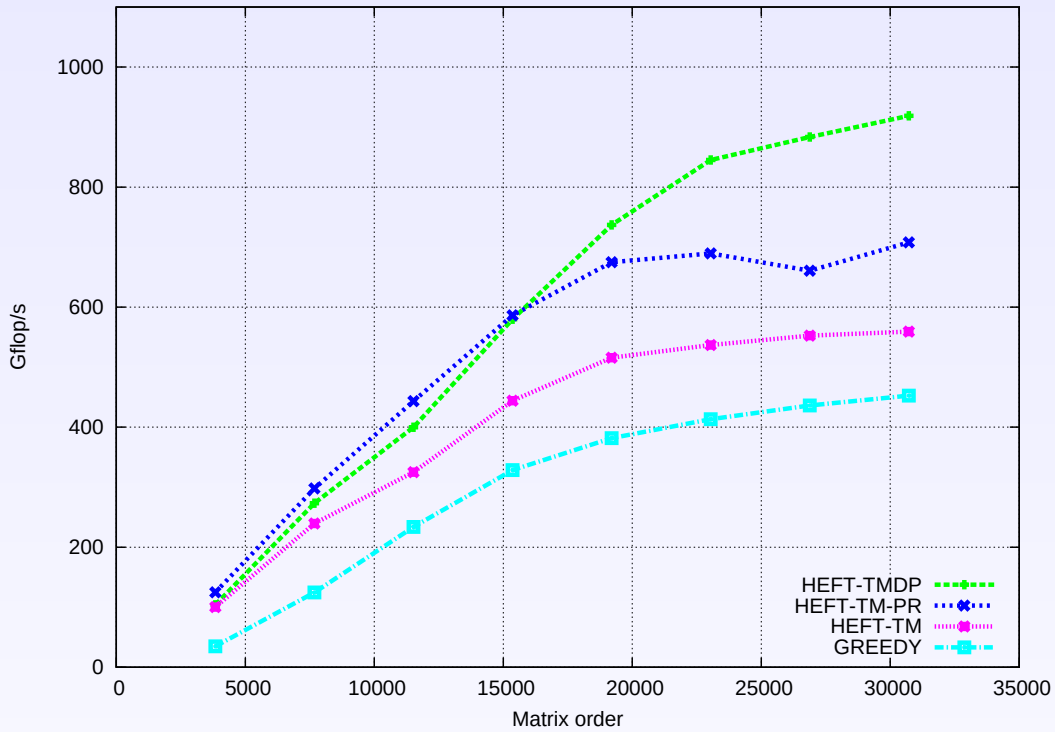

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the scheduling policy



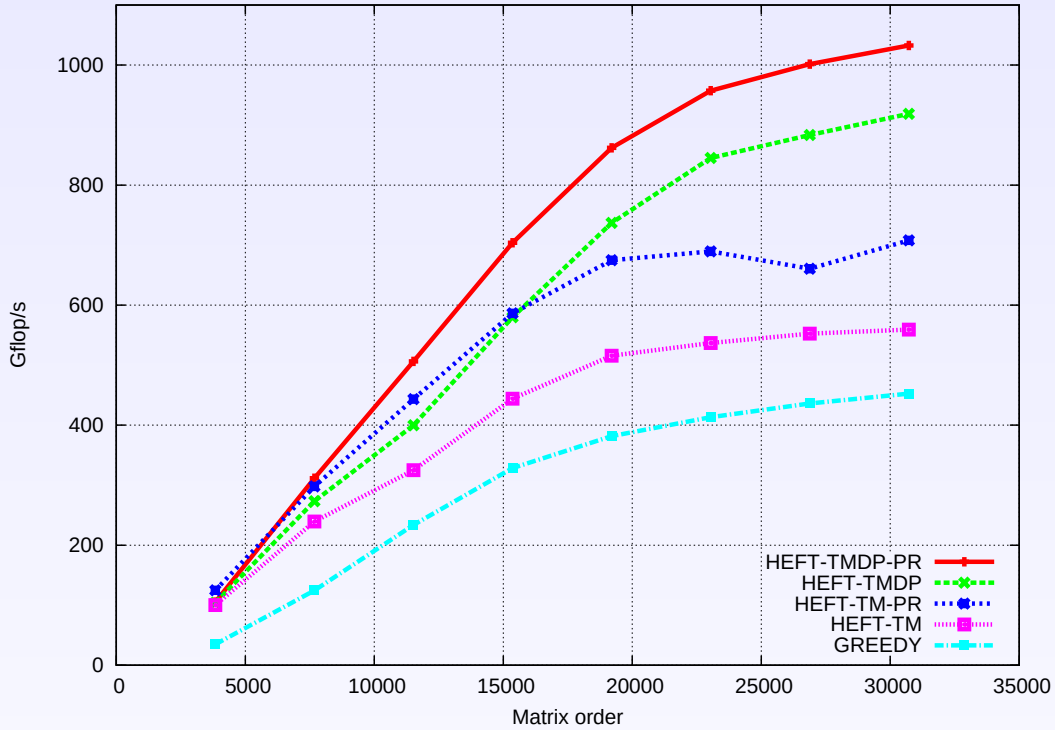

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the scheduling policy



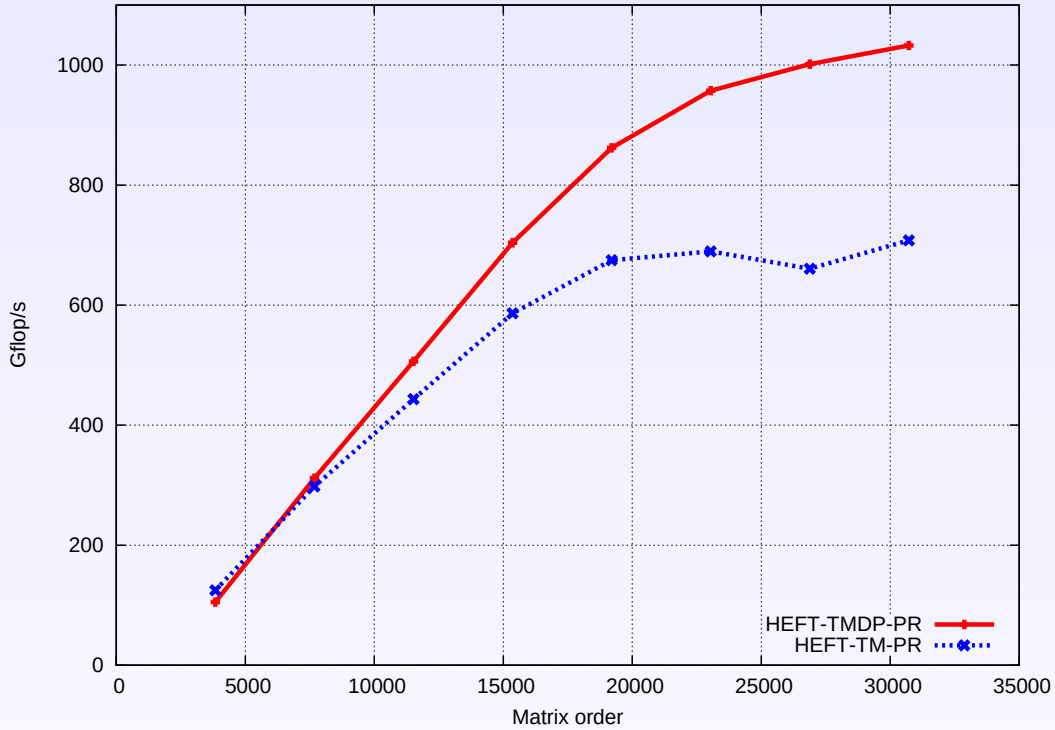

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the scheduling policy



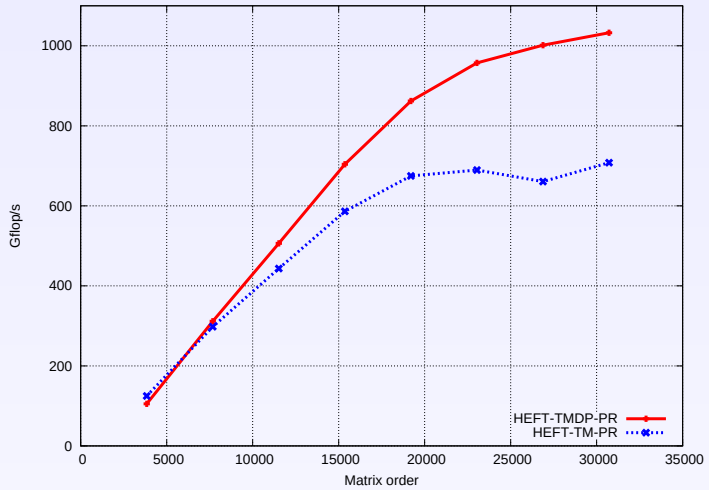

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the scheduling policy



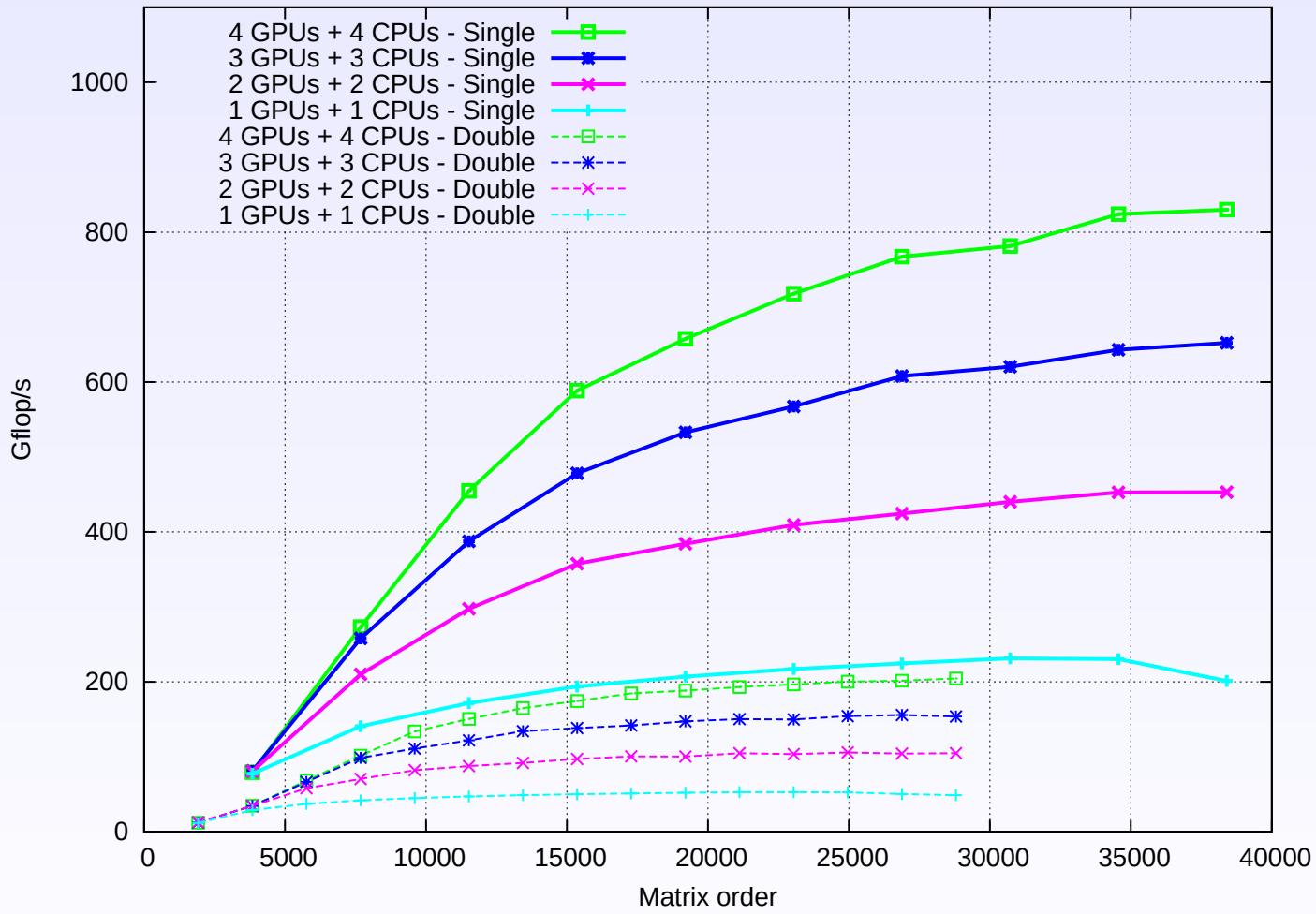

| Name | Policy description |
|---|---|
| **heft-tmdp-pr** | **heft-tmdp** with data PRefetch |
| **heft-tmdp** | **heft-tm** with remote Data Penalty ($\alpha T_{data\ transfert} + T_{computation}$) |
| **heft-tm-pr** | **heft-tm** with data PRefetch |
| **heft-tm** | HEFT based on Task duration Models ($T_{data\ transfert} + T_{computation}$) |
| **greedy** | Greedy policy |

# Impact of the Data Penalty on the total amount of data movement



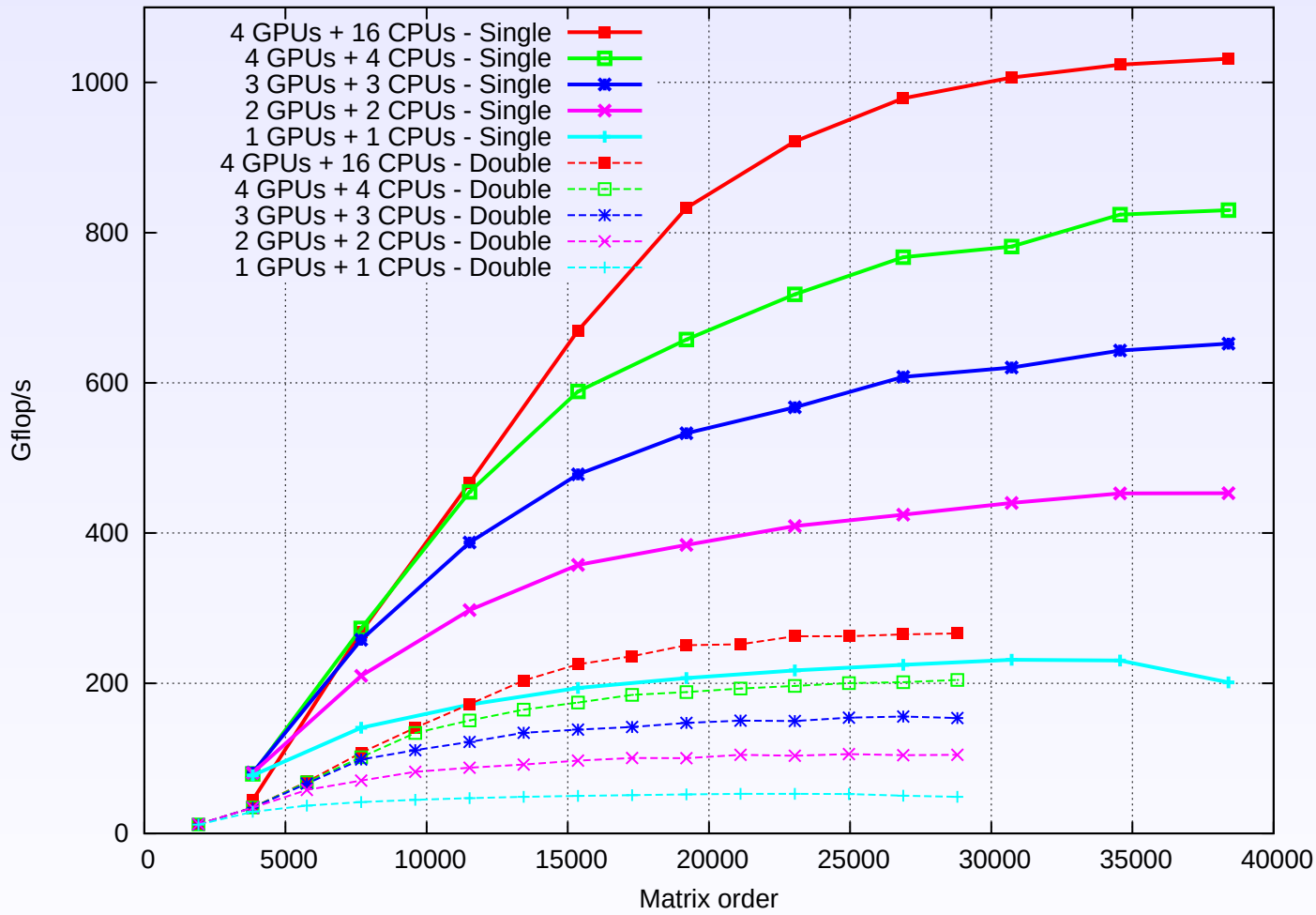

| Matrix order | 9600 | 24960 | 30720 | 34560 |
|---|---|---|---|---|
| **heft-tmdp-pr** | 1.9 GB | 16.3 GB | 25.4 GB | 41.6 GB |
| **heft-tm-pr** | 3.8 GB | 57.2 GB | 105.6 GB | 154.7 GB |

# Performance




+ 200Gflop/s but 12 cores = 150Gflop/s

# Performance




+ 200Gflop/s but 12 cores = 150Gflop/s

# Heterogeneity

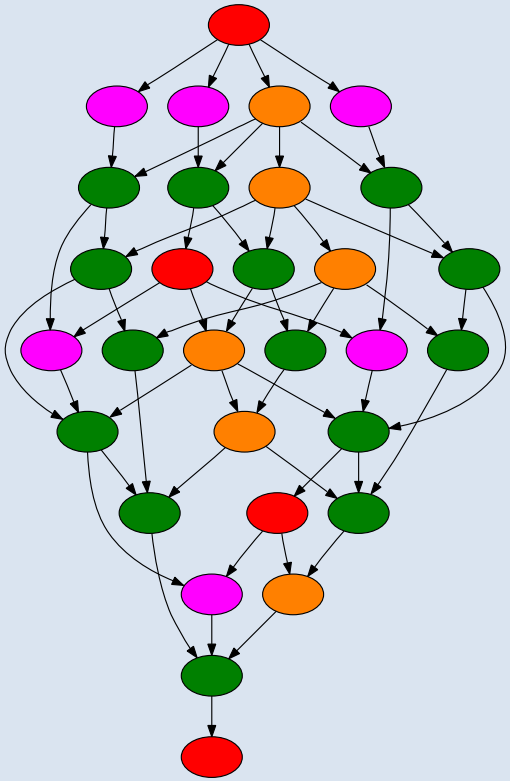
| Kernel | CPU | GPU | Speedup |
|---|---|---|---|
| `sgeqrt` | 9 Gflops | 60 Gflops | ≈6 |
| `stsqrt` | 12 Gflops | 67 Gflops | ≈6 |
| `sormqr` | 8.5 Gflops | 227 Gflops | ≈27 |
| `stsmqr` | 10 Gflops | 285 Gflops | ≈27 |

- Task distribution observed on StarPU:
  - `sgeqrt`: 20% of tasks on GPUs
  - `stsmqr`: 92.5% of tasks on GPUs
- Taking advantage of heterogeneity !
  - Only do what you are good for
  - Don't do what you are not good for

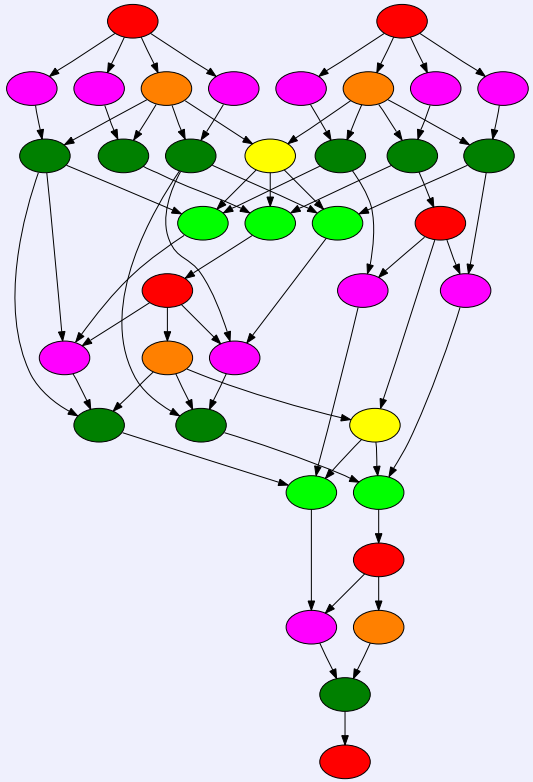
# Outline

# DAG of a 4x4 tile matrix

## 2D tile QR - flat tree

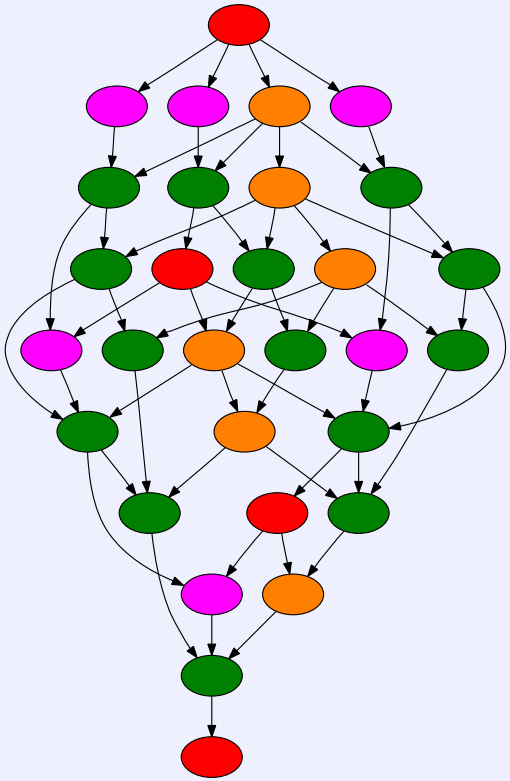


## 2D tile QR - hybrid binary/flat tree

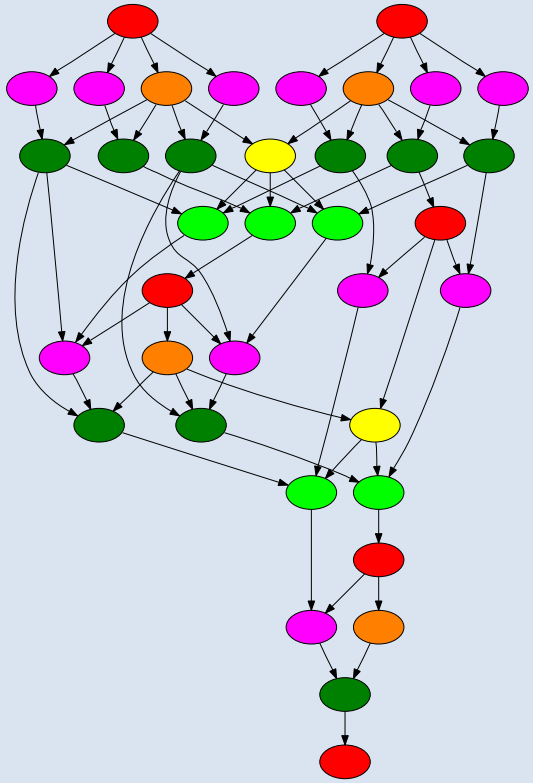


# DAG of a 4x4 tile matrix

## 2D tile QR - flat tree

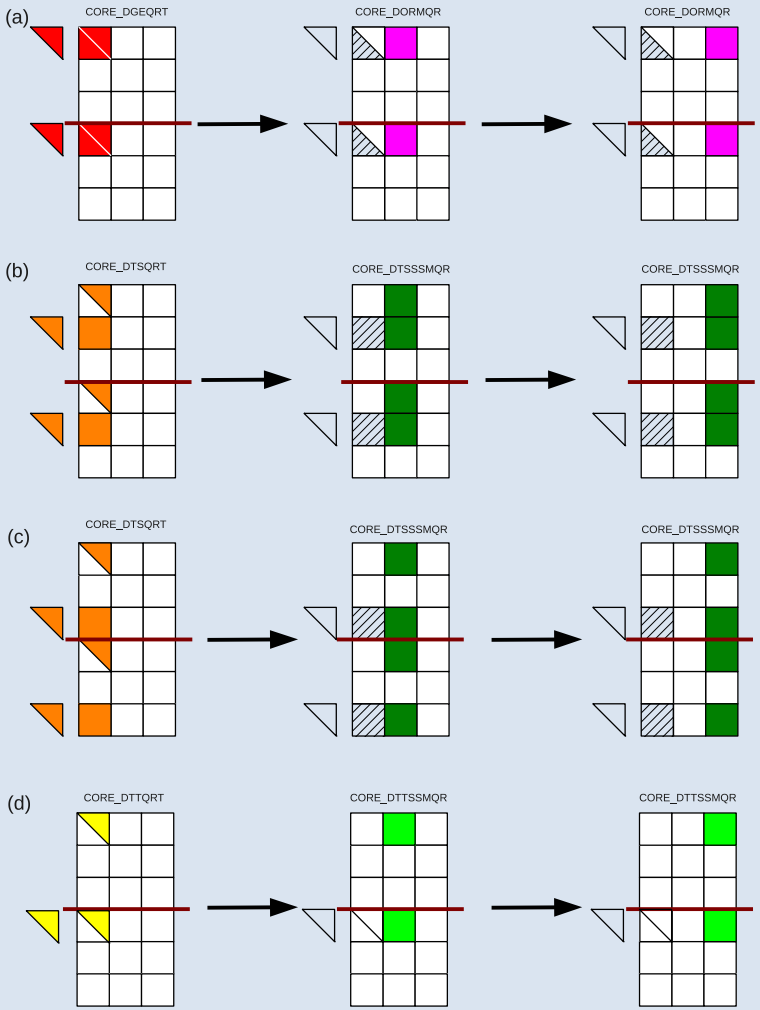


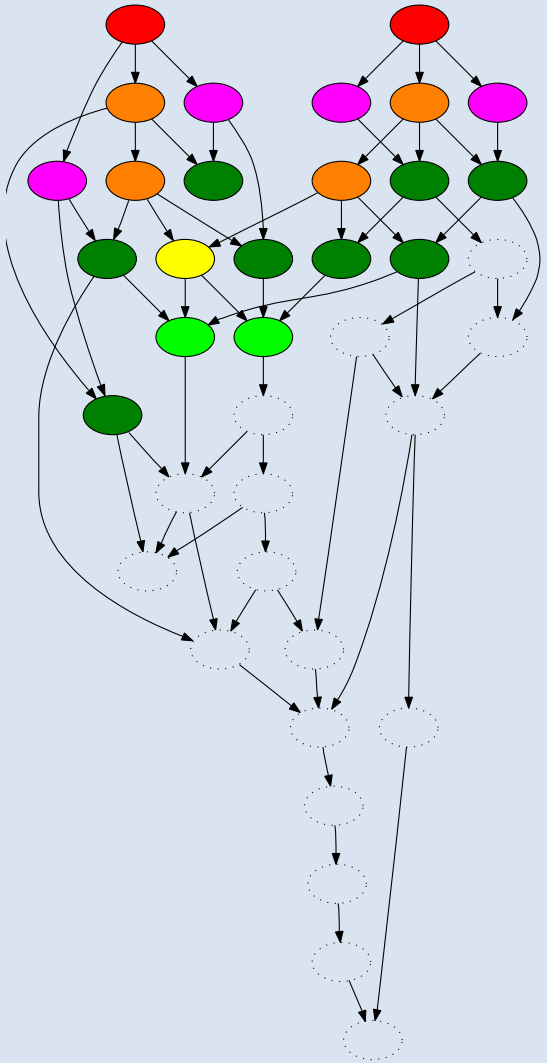
## 2D tile QR - hybrid binary/flat tree



# 2D tile QR - hybrid binary/flat tree

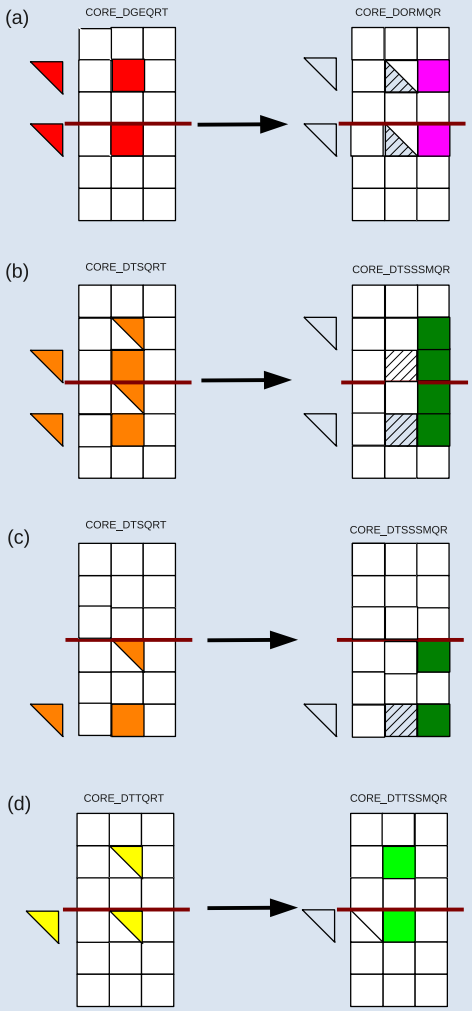
## First panel factorization and corresponding updates.

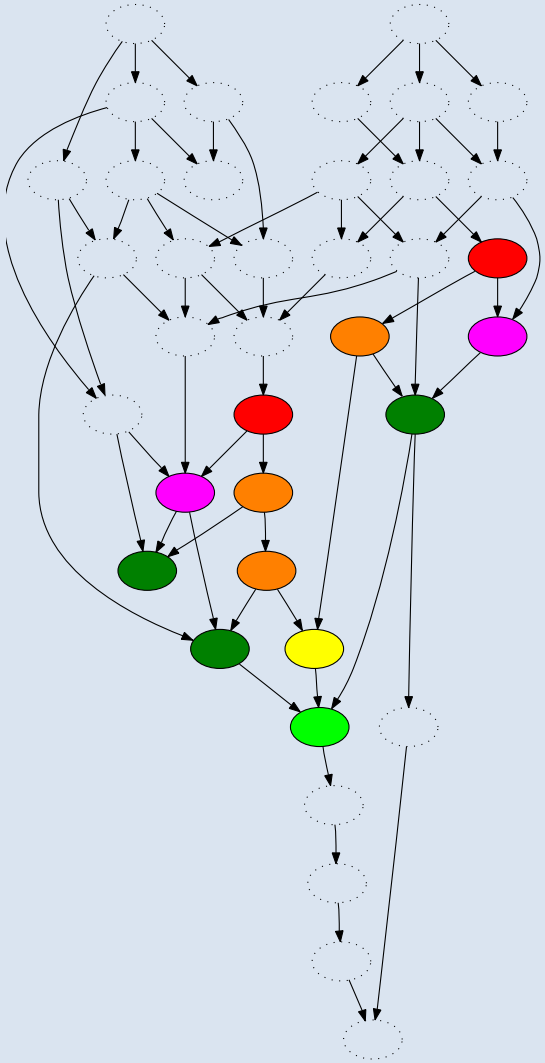


# 2D tile QR - hybrid binary/flat tree

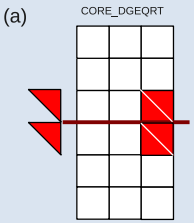
## Second panel factorization and corresponding updates.

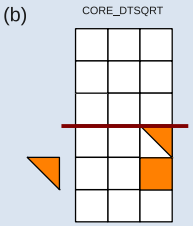


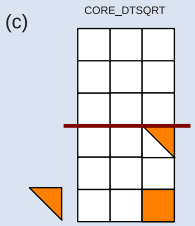



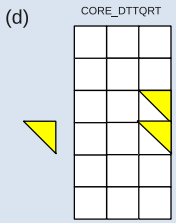
# 2D tile QR - hybrid binary/flat tree

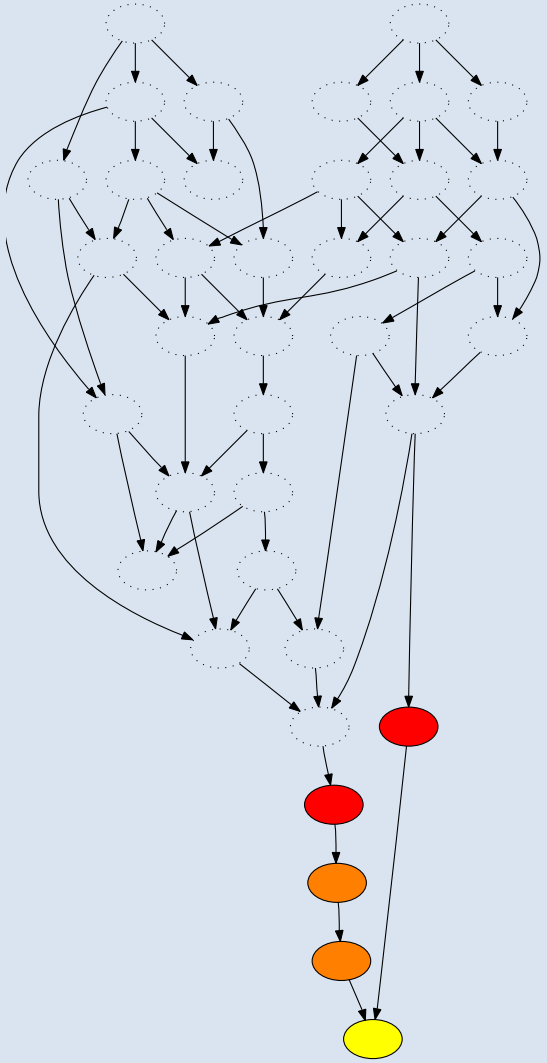
## Final panel factorization.



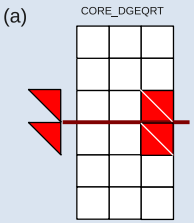



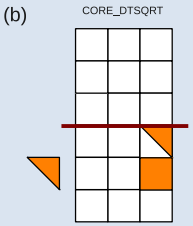



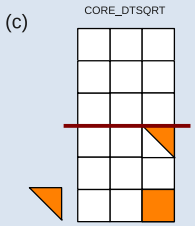



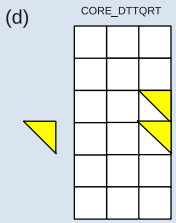
# 2D tile QR - hybrid binary/flat tree

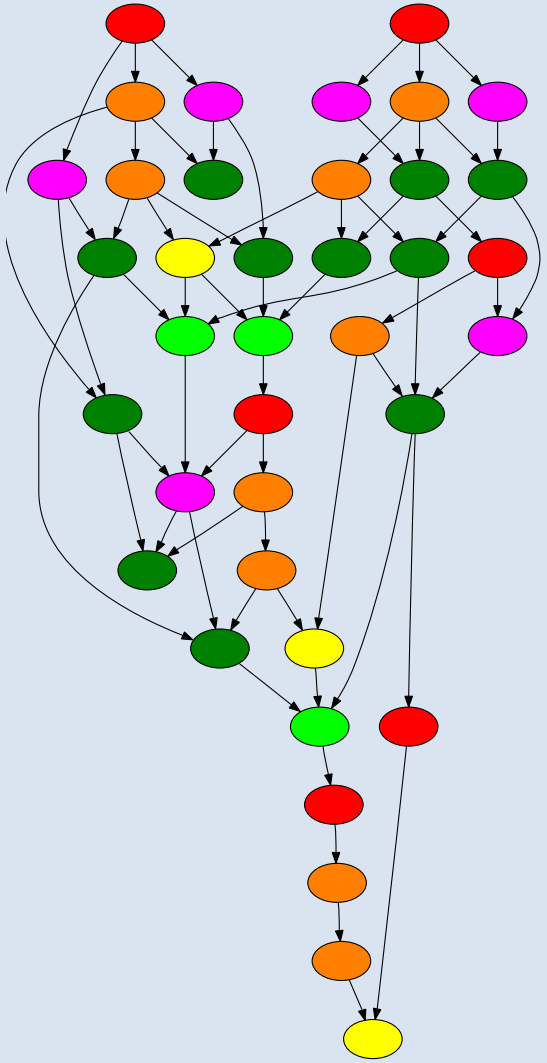
## Final panel factorization.



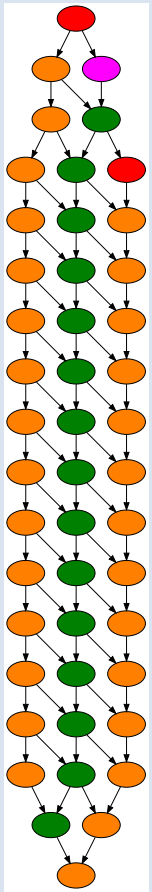



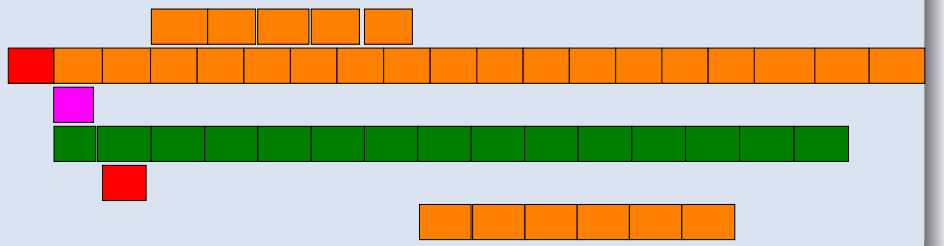



# 16x2 tile matrix - 1 domain (flat tree)

## DAG

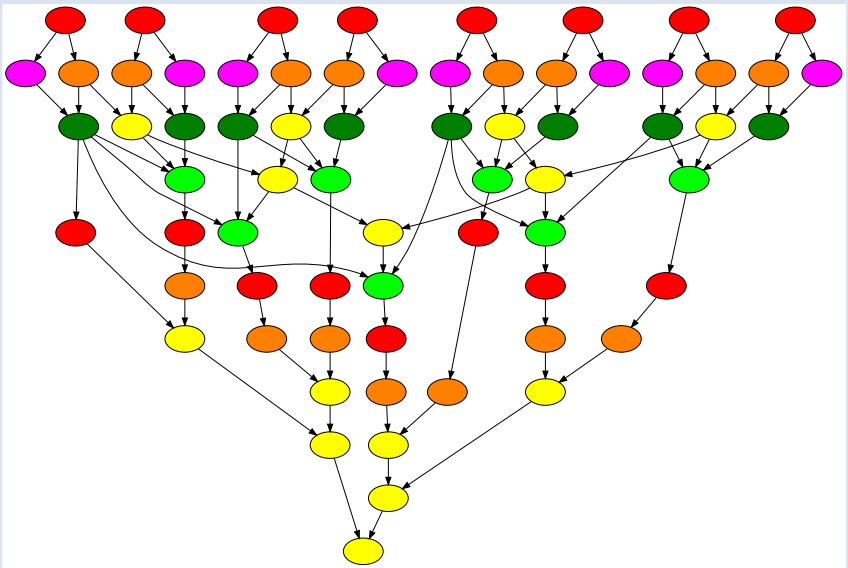


## Traces (8 cores)

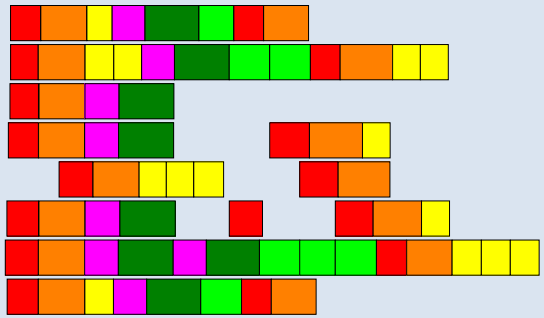


# 16x2 tile matrix - 8 domains (hybrid tree)

## DAG

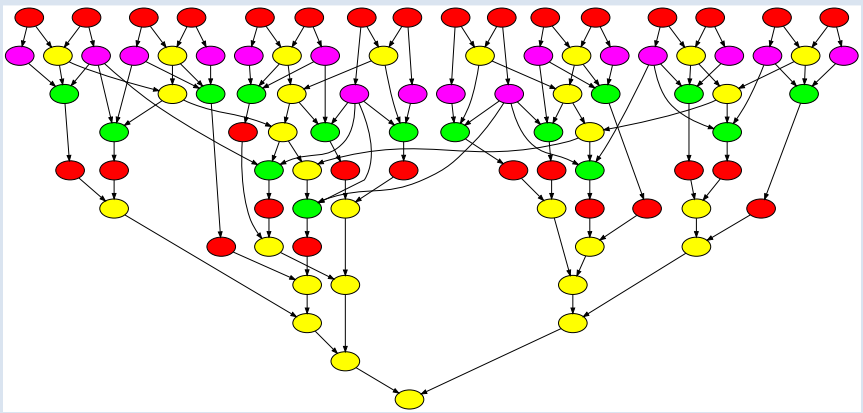


## Traces (8 cores)

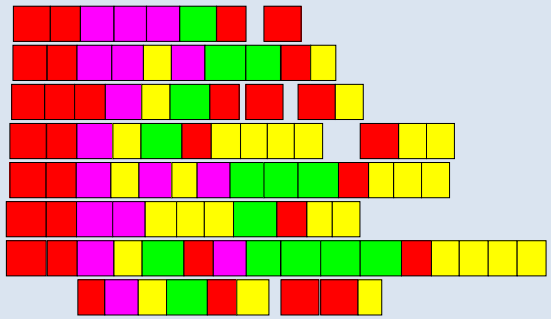


# 16x2 tile matrix - 16 domains (binary tree)

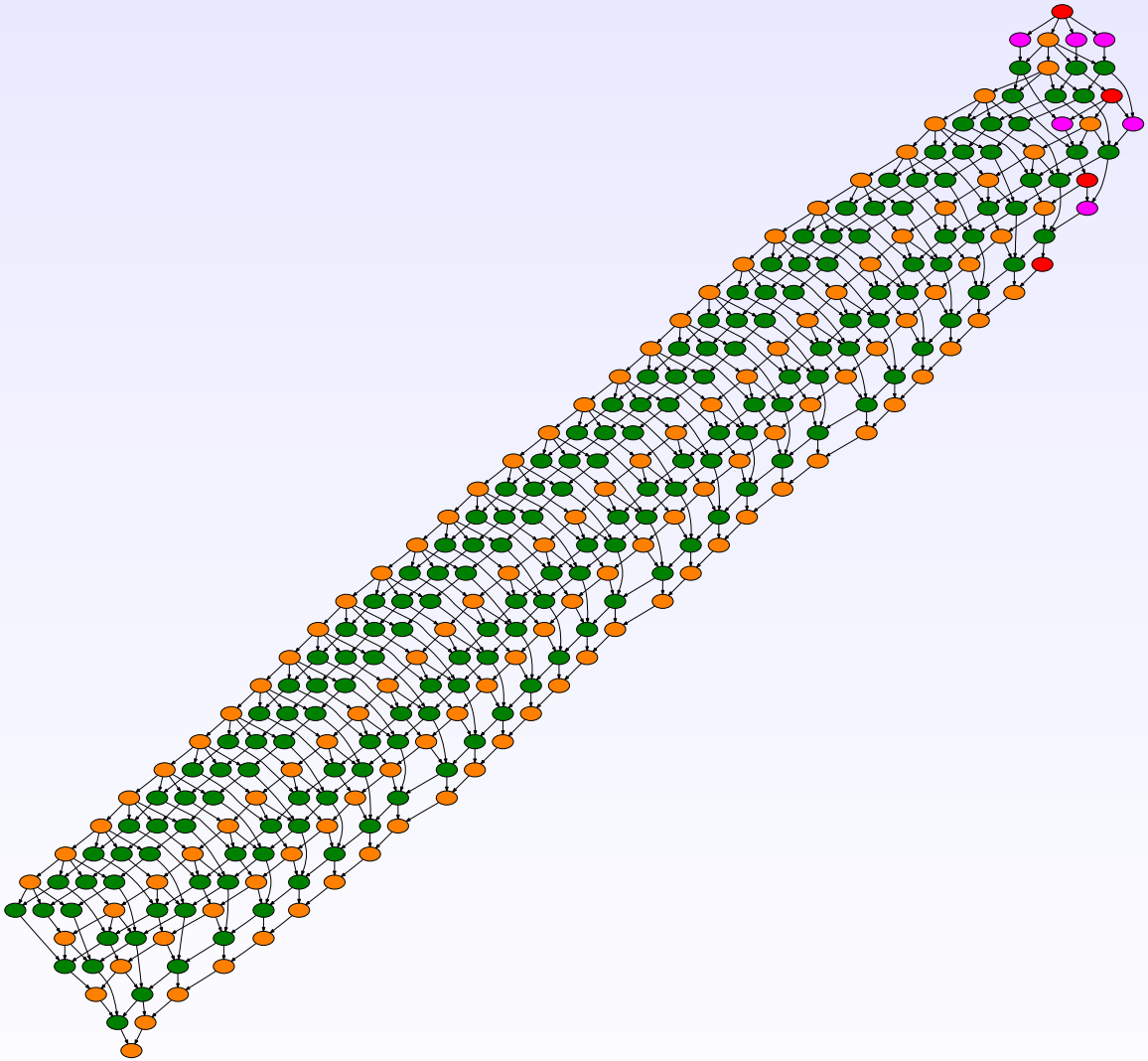
## DAG



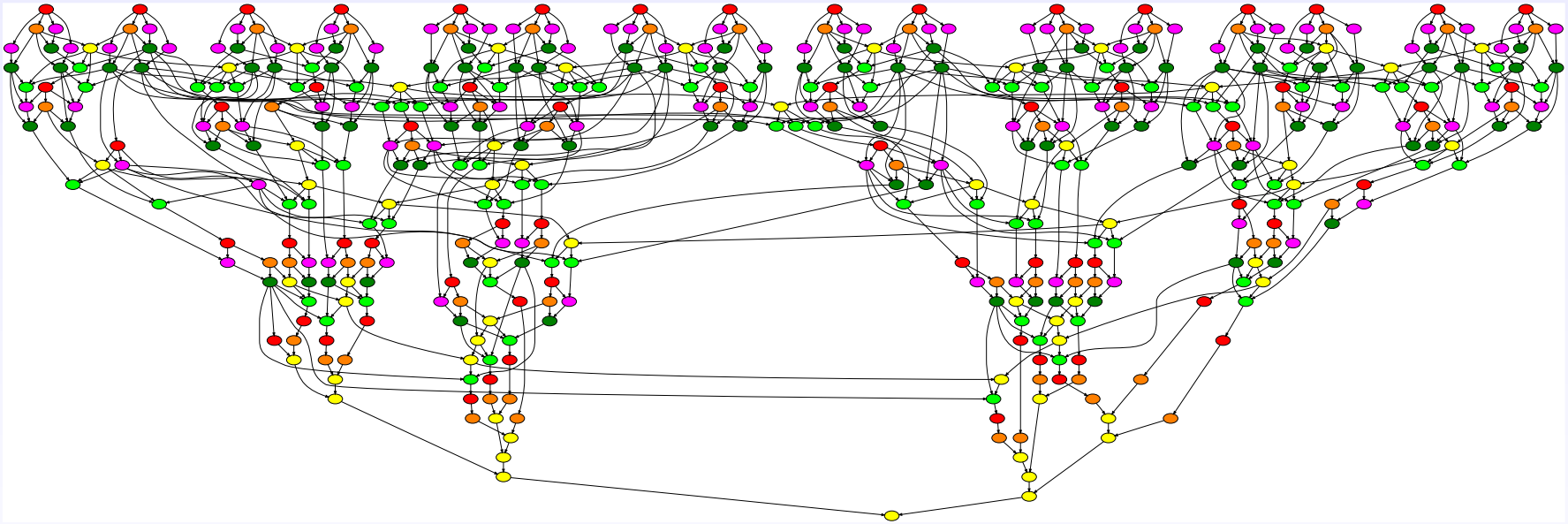
## Traces (8 cores)



# 32x4 tile matrix - 1 domain (flat tree)

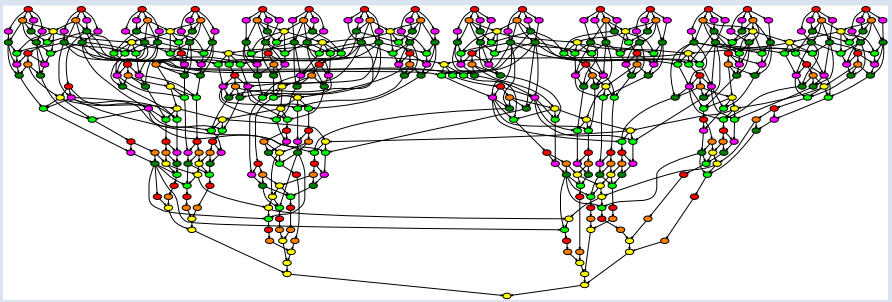


# 32x4 tile matrix - 16 domains (hybrid tree)

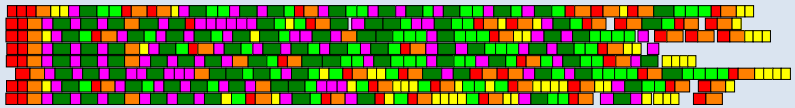


# 32x4 tile matrix - 16 domains (hybrid tree)

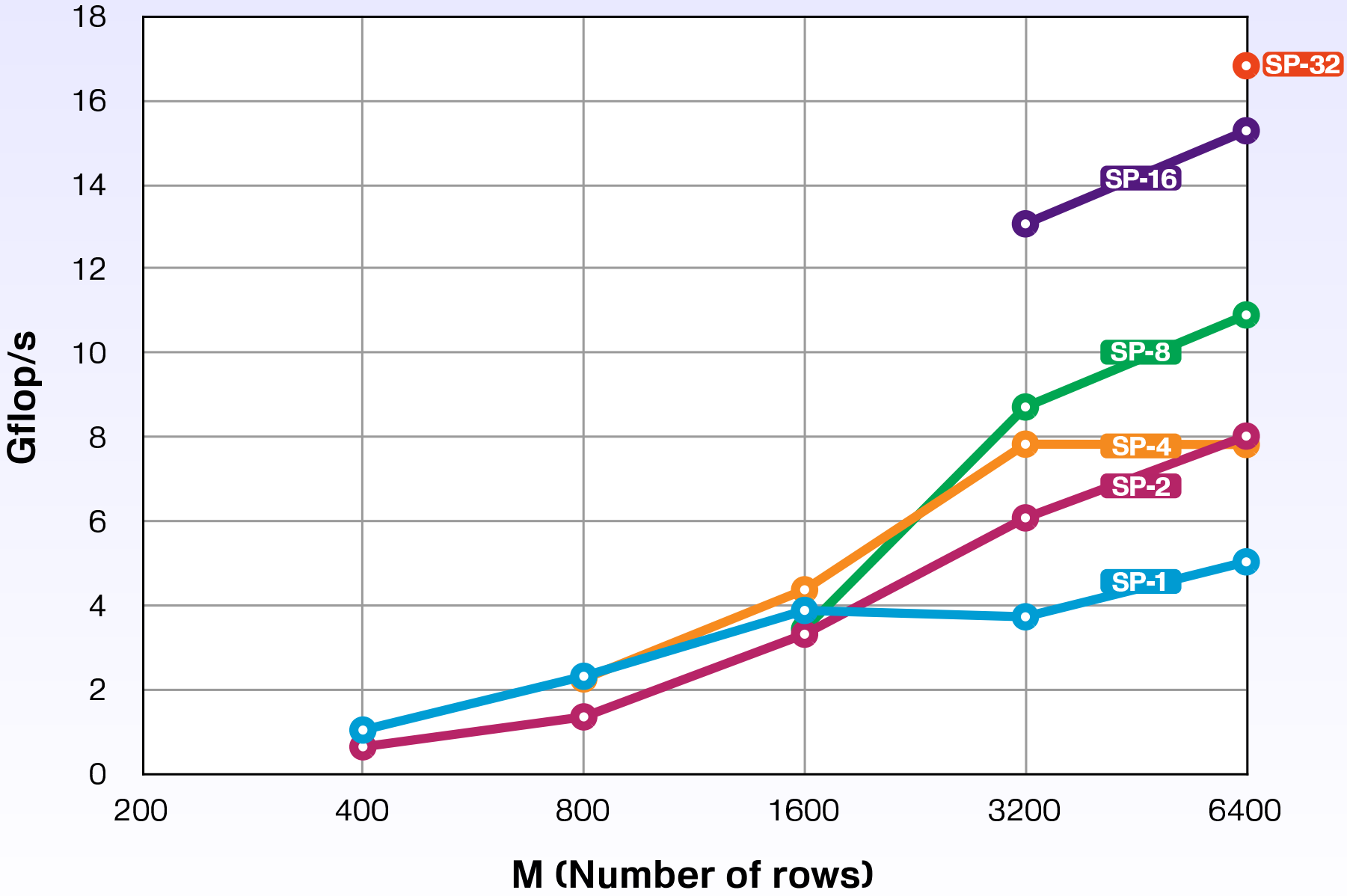
## DAG



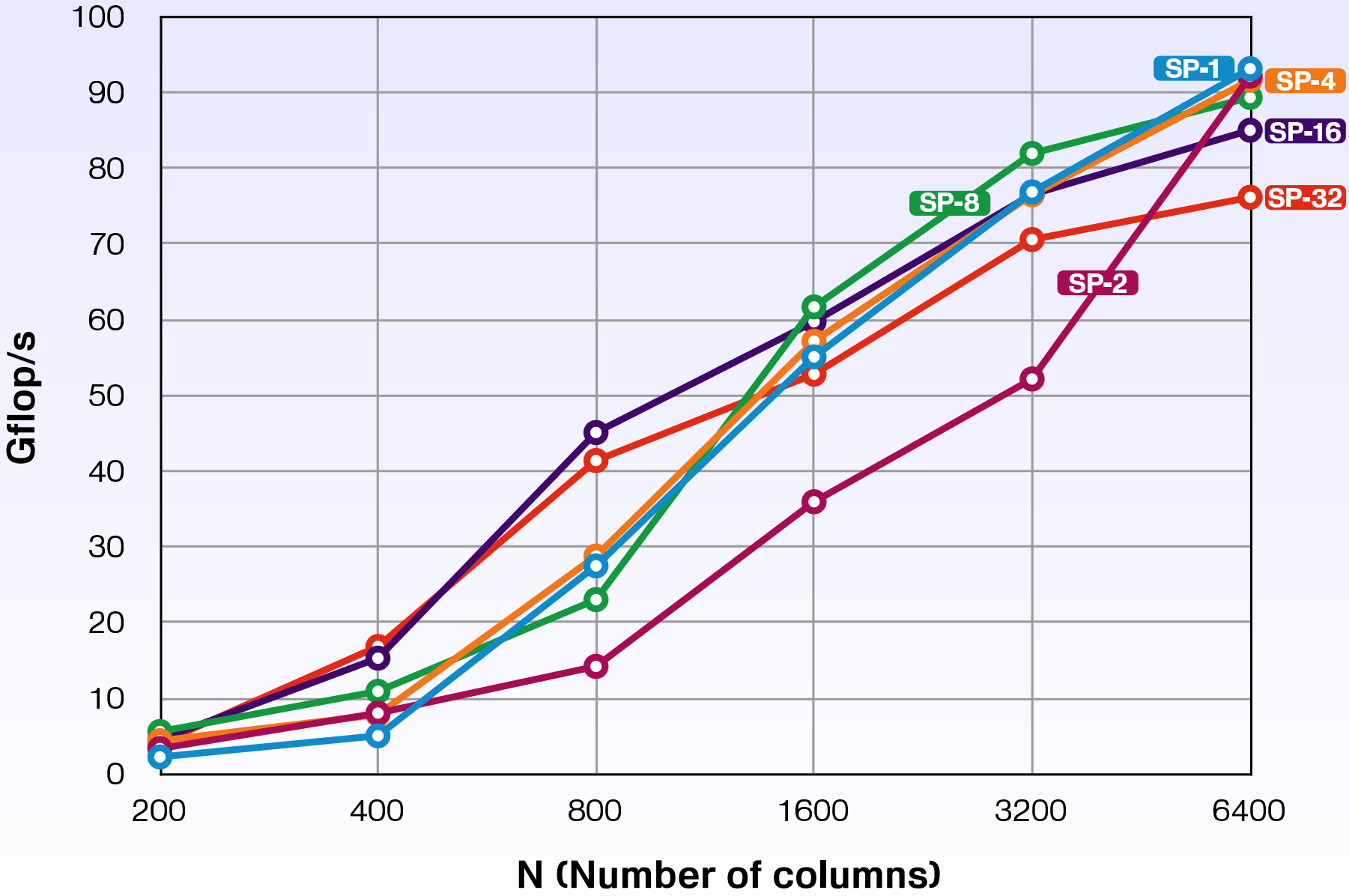
## Traces (8 cores)



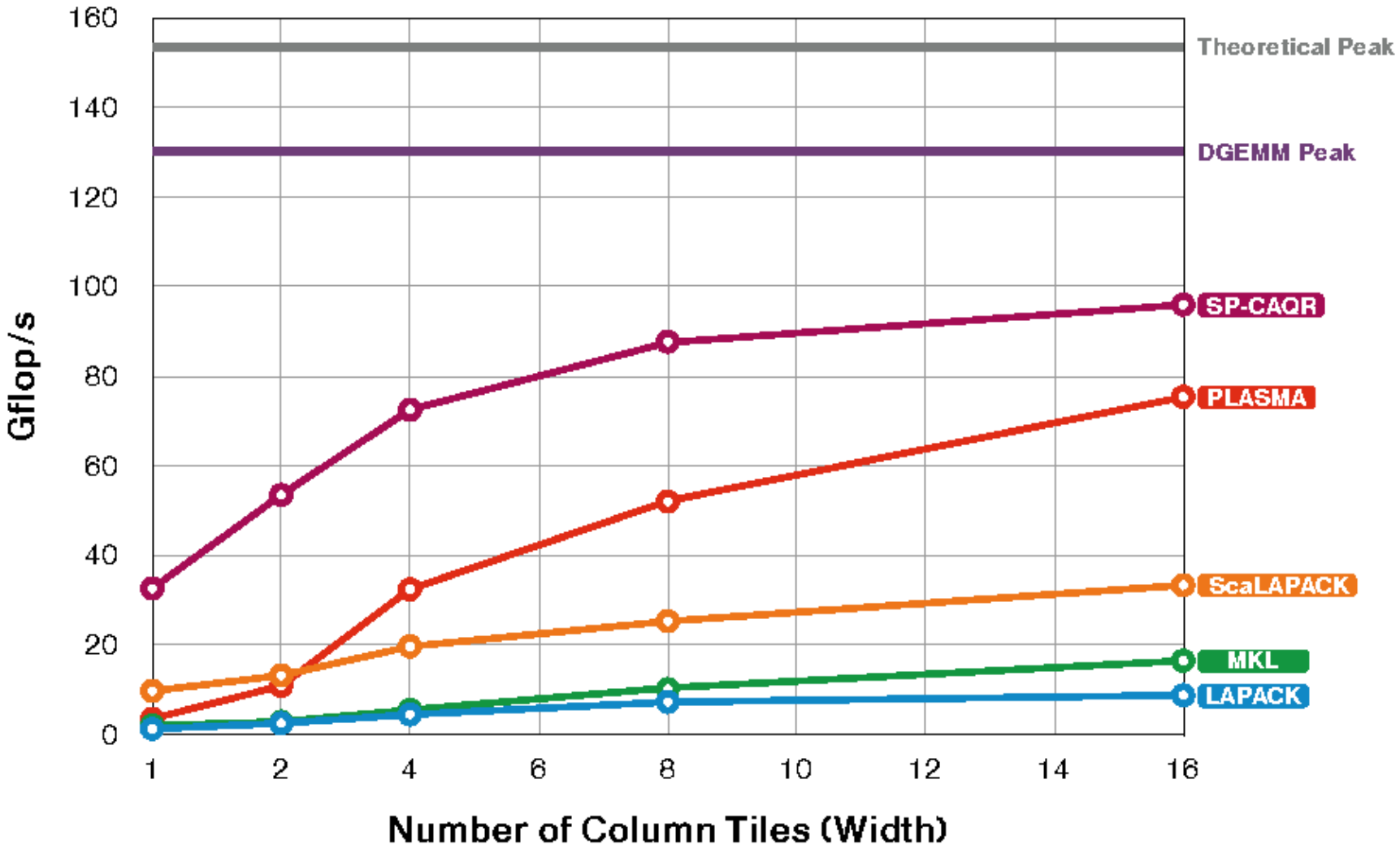
# $N = 400$ - 16 cores



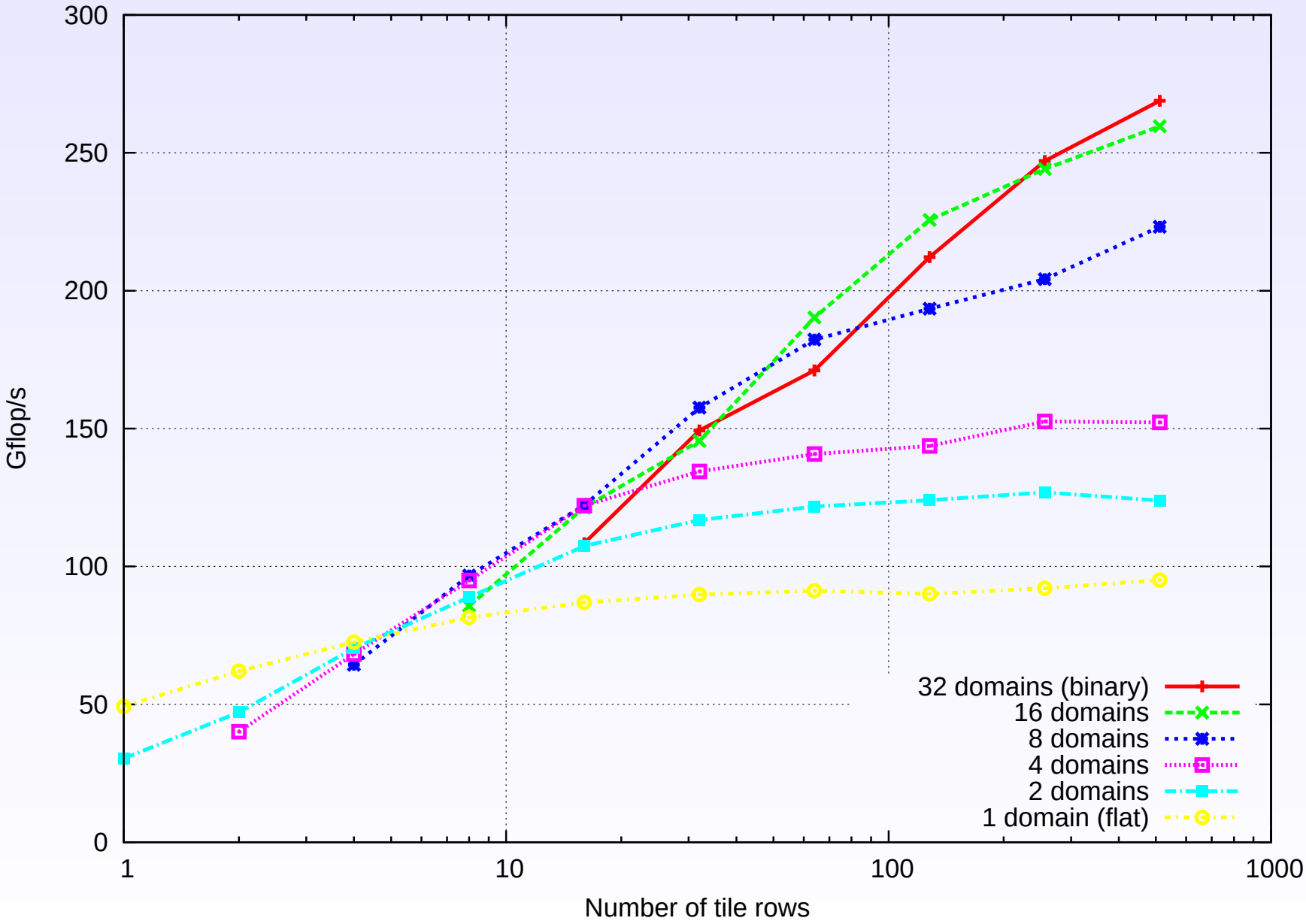
# $M = 6400$ - 16 cores



# $M = 51200 - 200 \leq N \leq 3200$

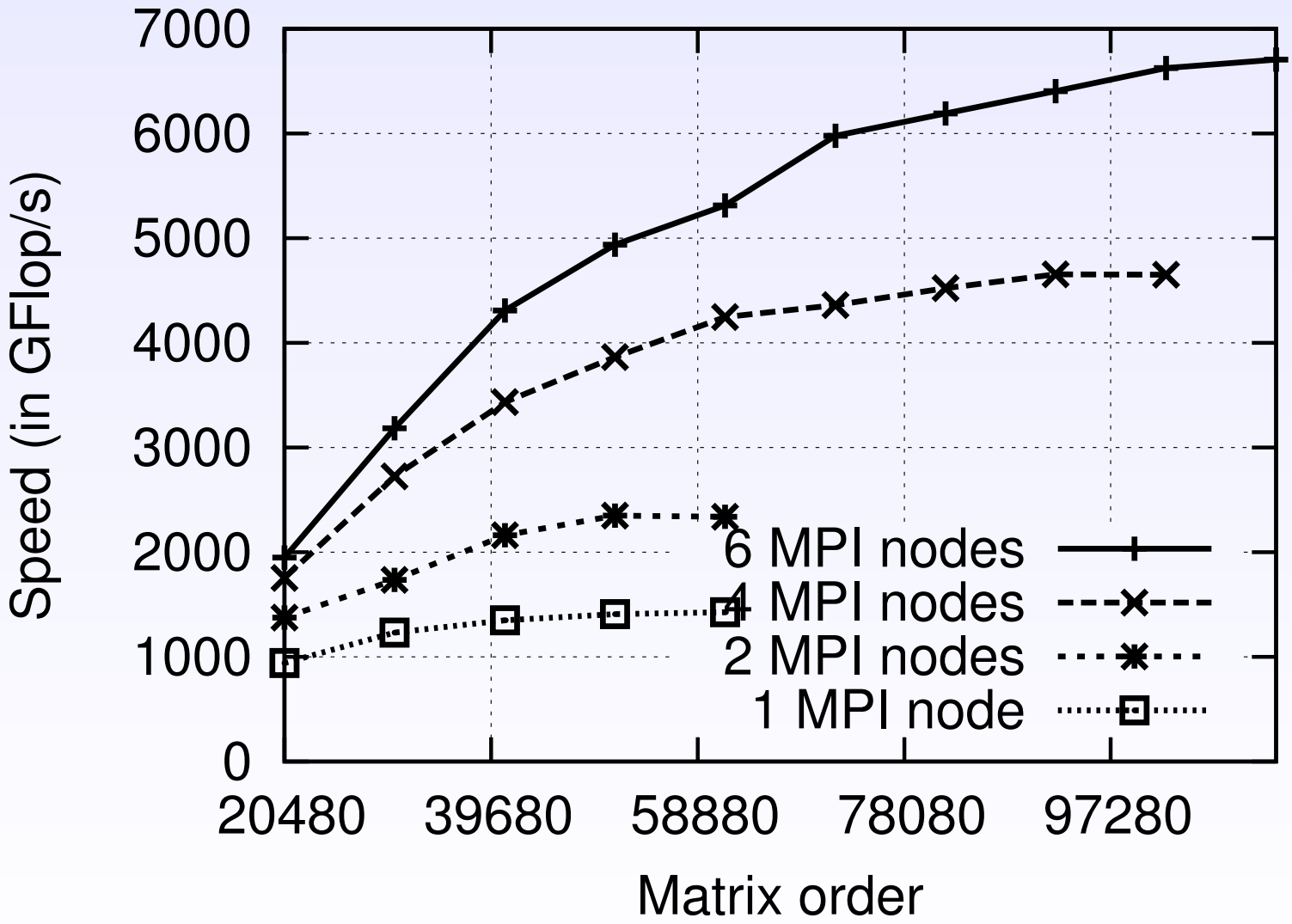


# Heterogeneous platform - double precision - N = 2*960

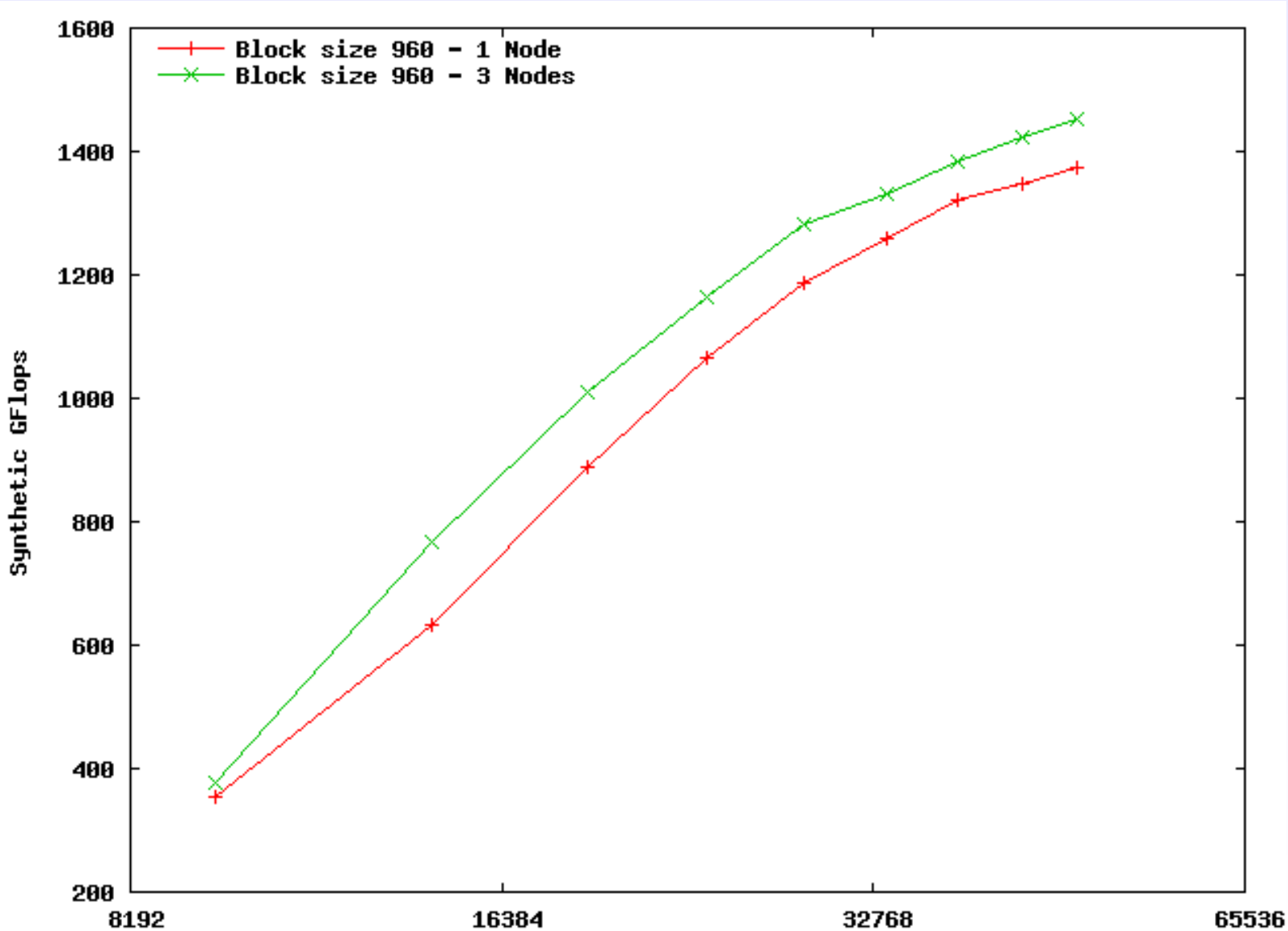


# Outline

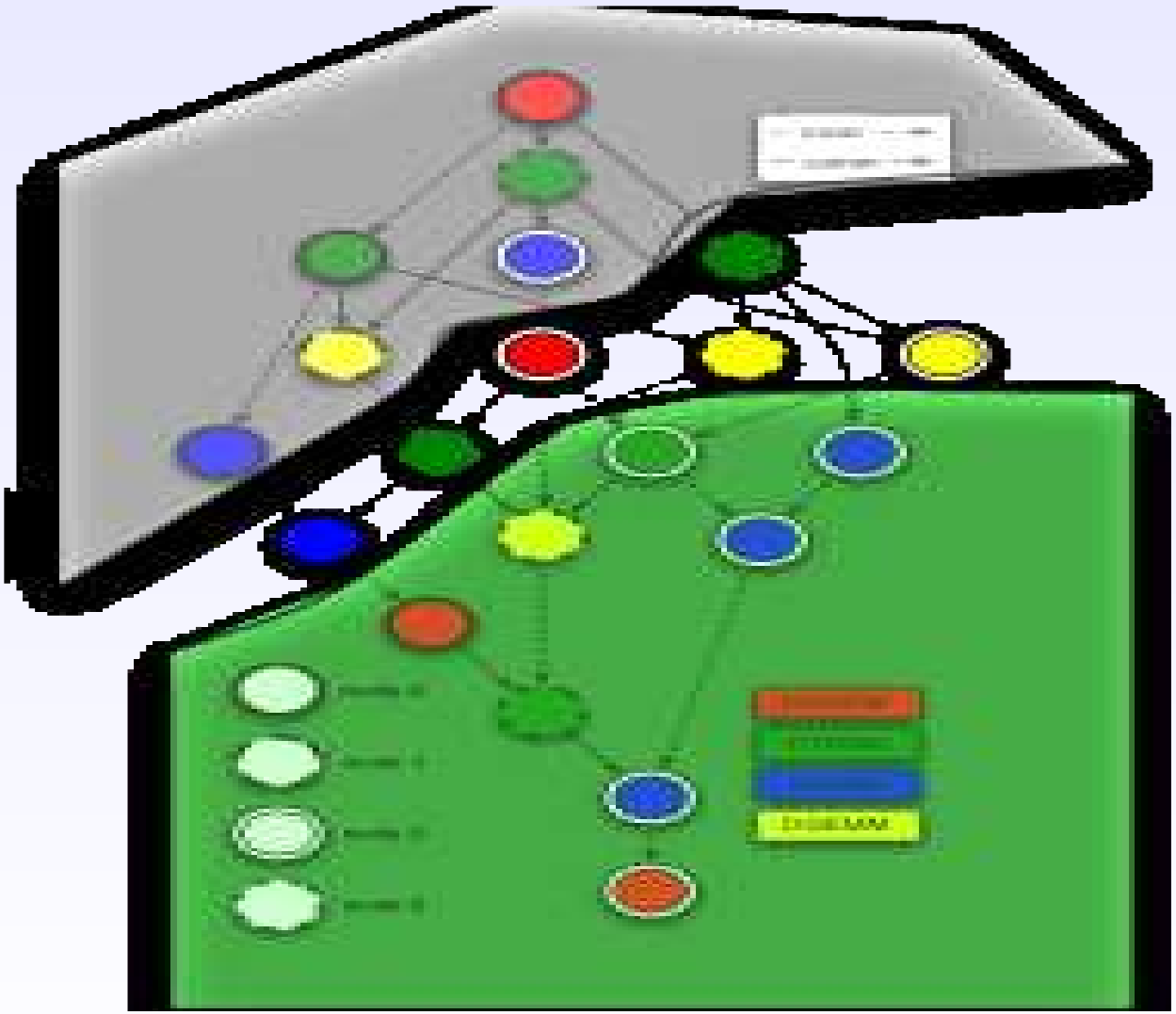
# Scalability - Cholesky
## 6 nodes ; 3 GPUs and 12 cores per node



# Scalability - Cholesky
# Total of 3 GPUs and 12 cores



# Scalability: towards exascale ?



# Conclusion

## Conclusion

- three-layers paradigm;
- productivity and performance portability;
- Cholesky, (CA-)QR and LU solvers;
- to be released into the MAGMA library.

## Perspectives (and additional layers)

- high-level complex routines (eigensolvers, sparse solvers, ...)
- scalable runtime;
- high-performance kernels;
- Autotuning framework;
- Clever scheduling algorithms;
- ...

Thanks!

# Conclusion

## Conclusion

- three-layers paradigm;
- productivity and performance portability;
- Cholesky, (CA-)QR and LU solvers;
- to be released into the MAGMA library.

## Perspectives (and additional layers)

- high-level complex routines (eigensolvers, sparse solvers, ...)
- scalable runtime;
- high-performance kernels;
- Autotuning framework;
- Clever scheduling algorithms;
- ...

Thanks!

# Conclusion

## Conclusion

- three-layers paradigm;
- productivity and performance portability;
- Cholesky, (CA-)QR and LU solvers;
- to be released into the MAGMA library.

## Perspectives (and additional layers)

- high-level complex routines (eigensolvers, sparse solvers, ...)
- scalable runtime;
- high-performance kernels;
- Autotuning framework;
- Clever scheduling algorithms;
- ...

Thanks!

# Conclusion

## Conclusion

- three-layers paradigm;
- productivity and performance portability;
- Cholesky, (CA-)QR and LU solvers;
- to be released into the MAGMA library.

## Perspectives (and additional layers)

- high-level complex routines (eigensolvers, sparse solvers, ...)
- scalable runtime;
- high-performance kernels;
- Autotuning framework;
- Clever scheduling algorithms;
- ...

Thanks!

# Conclusion

## Conclusion

- three-layers paradigm;
- productivity and performance portability;
- Cholesky, (CA-)QR and LU solvers;
- to be released into the MAGMA library.

## Perspectives (and additional layers)

- high-level complex routines (eigensolvers, sparse solvers, ...)
- scalable runtime;
- high-performance kernels;
- Autotuning framework;
- Clever scheduling algorithms;
- ...

Thanks!