CS 594 — Scientific Computing for Engineers — Spring 2018

Dense Linear Algebra

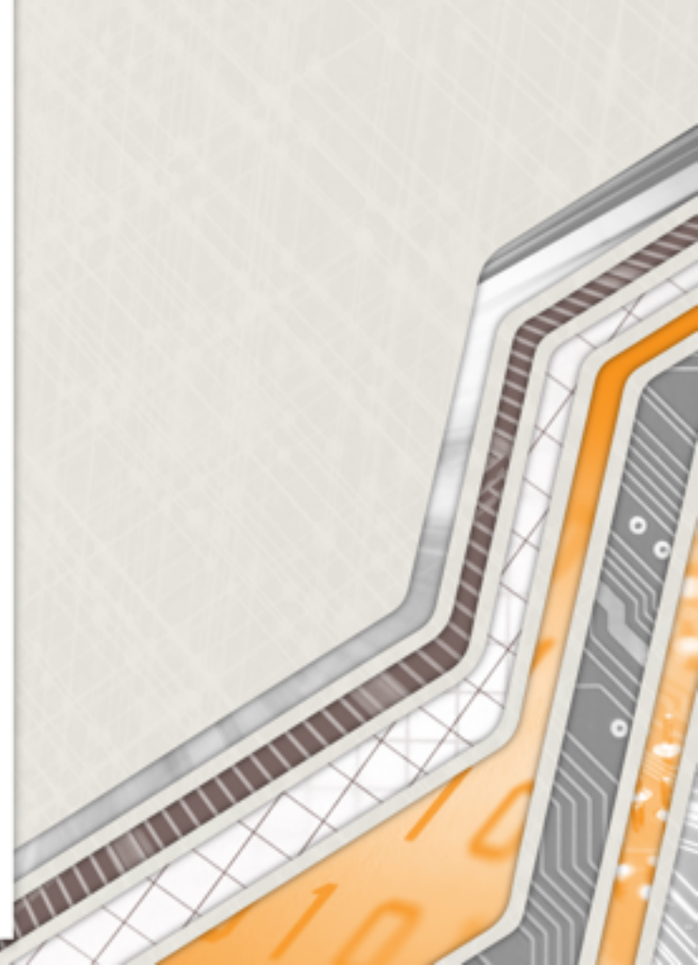
Mark Gates

mgates3@icl.utk.edu

http://www.icl.utk.edu/~mgates3/

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE



Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

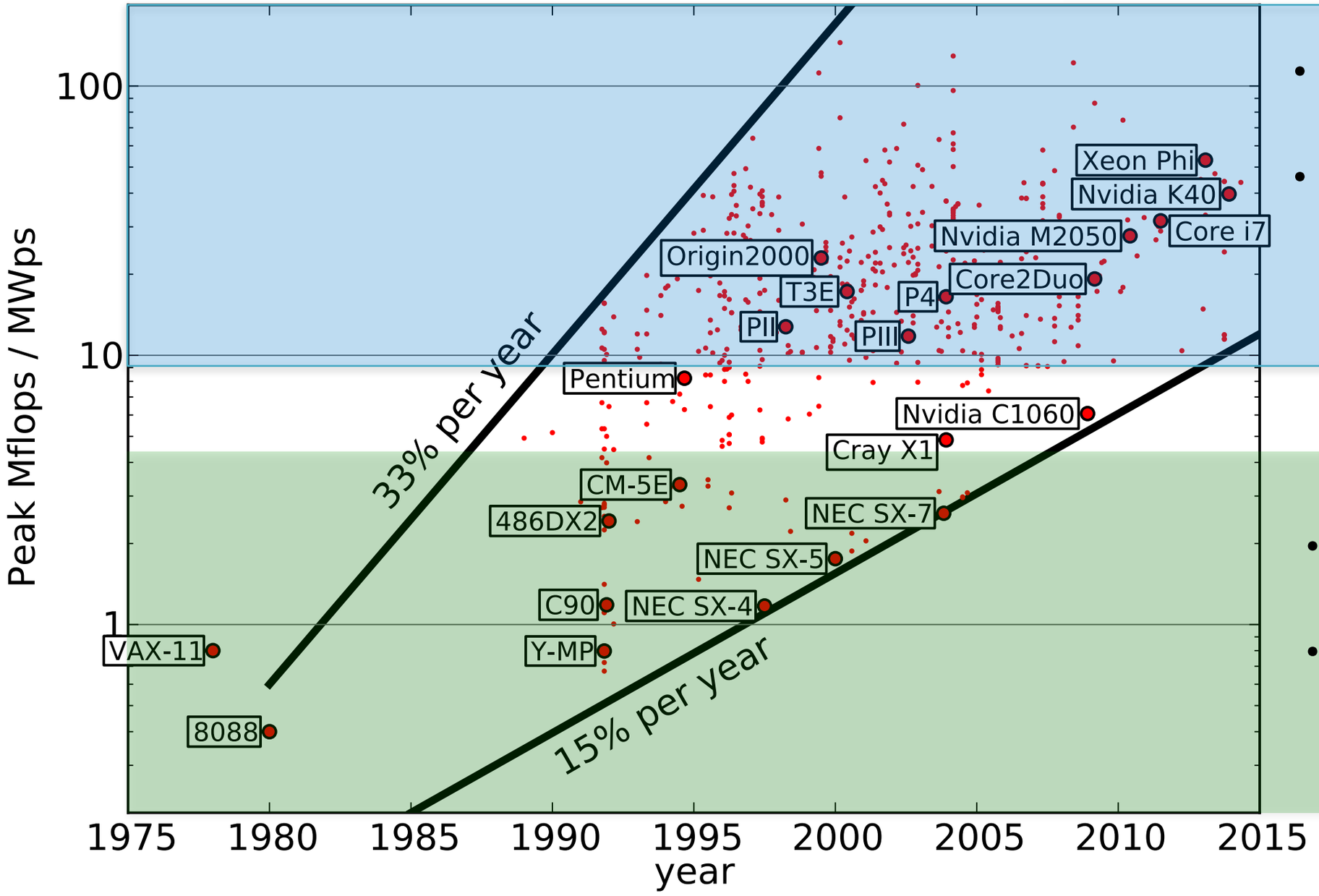
Memory hierarchy

		Examples (Haswell)
Fastest Most expensive Smallest capacity	CPU Registers	168 integer 168 float
Fast Expensive Small capacity	CPU cache L1 Level 1 cache L2 Level 2 cache L3 Level 3 cache	32 KiB L1 per core 256 KiB L2 per core 1.5 MiB L3 per core
Modest speed Modest cost Modest capacity	Random Access Memory (RAM) SDRAM, DDR, GDDR, HBM, etc.	64 GiB
Slow Cheap Large capacity	Solid State Drive (SSD, Flash) Burst buffers, Non-volatile Memory, Files	500 GiB
Slowest Cheapest Largest capacity	Mechanical Hard Drive (HDD) Virtual Memory, Files	4 TiB

Adapted from illustration by Ryan Leng

Compute vs. memory speed

- Machine balance (# flops per RAM access)



- Flops “free,” memory expensive
- Good for dense, BLAS-3 operations (matrix multiply)

- Flops & memory access balanced
- Good for sparse & vector operations

Data from Stream benchmark (McCalpin) and vendor information pages.

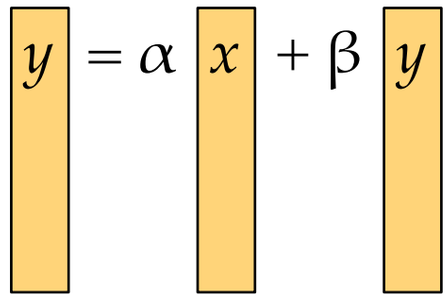
Outline

- Legacy Software
  - **BLAS**
  - LINPACK
  - LAPACK
  - ScaLAPACK

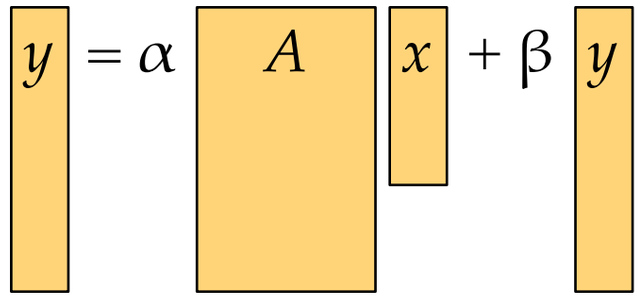
- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

BLAS: Basic Linear Algebra Subroutines

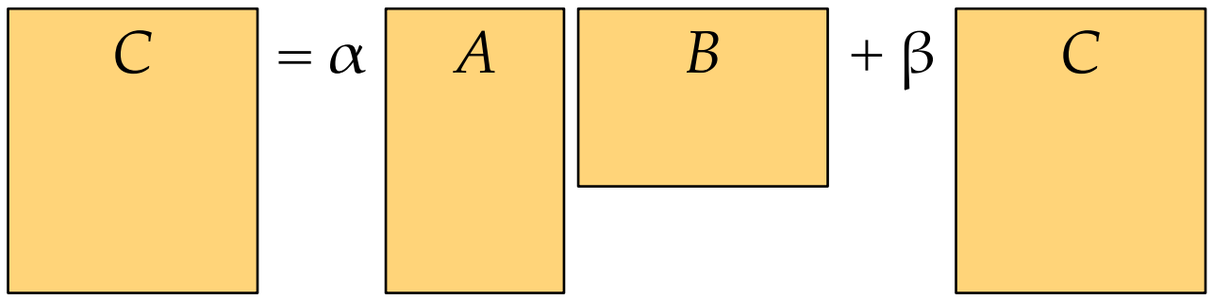
- Level 1 BLAS — vector operations
  - $O(n)$ data and flops (floating point operations)
  - Memory bound:
    $O(1)$ flops per memory access



- Level 2 BLAS — matrix-vector operations
  - $O(n^2)$ data and flops
  - Memory bound:
    $O(1)$ flops per memory access

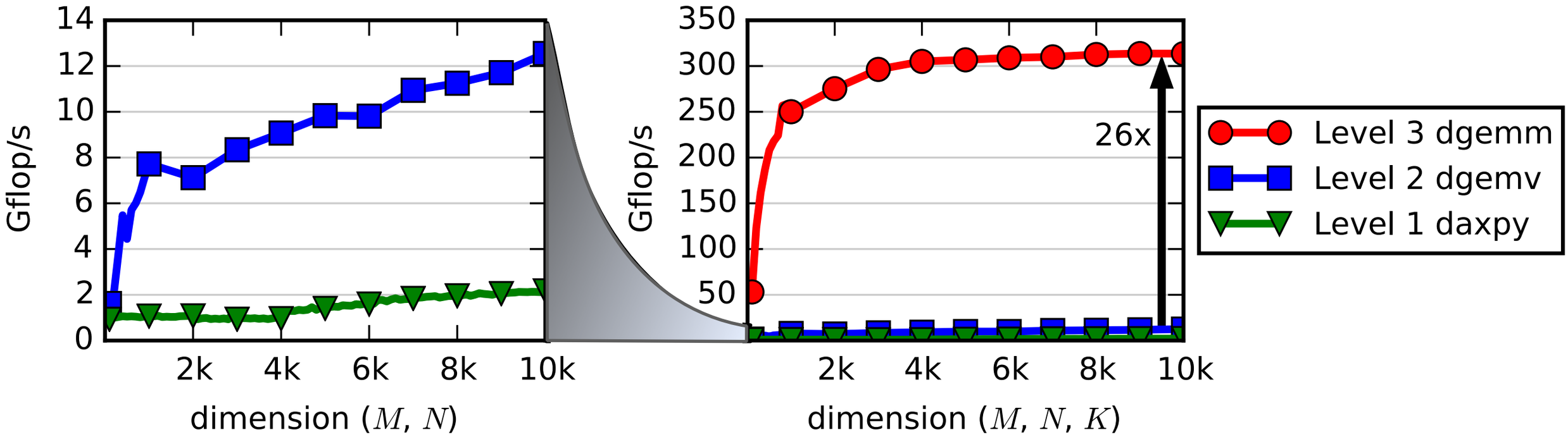


- Level 3 BLAS — matrix-matrix operations
  - $O(n^2)$ data, $O(n^3)$ flops
  - Surface-to-volume effect
  - Compute bound:
    $O(n)$ flops per memory access



BLAS: Basic Linear Algebra Subroutines

- Intel Sandy Bridge (2 socket E5-2670)
  - Peak 333 Gflop/s = 2.6 GHz * 16 cores * 8 double-precision flops/cycle †
  - Max memory bandwidth 51 GB/s ‡



† http://stackoverflow.com/questions/15655835/flops-per-cycle-for-sandy-bridge-and-haswell-sse2-avx-avx2

‡ http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI

Memory bandwidth

- Intel Ark
  - https://ark.intel.com/
  - Search for model number (e.g., “E5-2697 v4” for Broadwell, see /proc/cpuinfo)

```
prompt> less /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz
...
cache size      : 46080 KB
siblings        : 18
physical id     : 0
core id         : 1
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx
est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes
xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm intel_pt tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap
xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local
...
```

siblings = # hyperthreads per socket (see ark)

each physical id / core id combination is unique core

supports SSE and AVX vector instructions

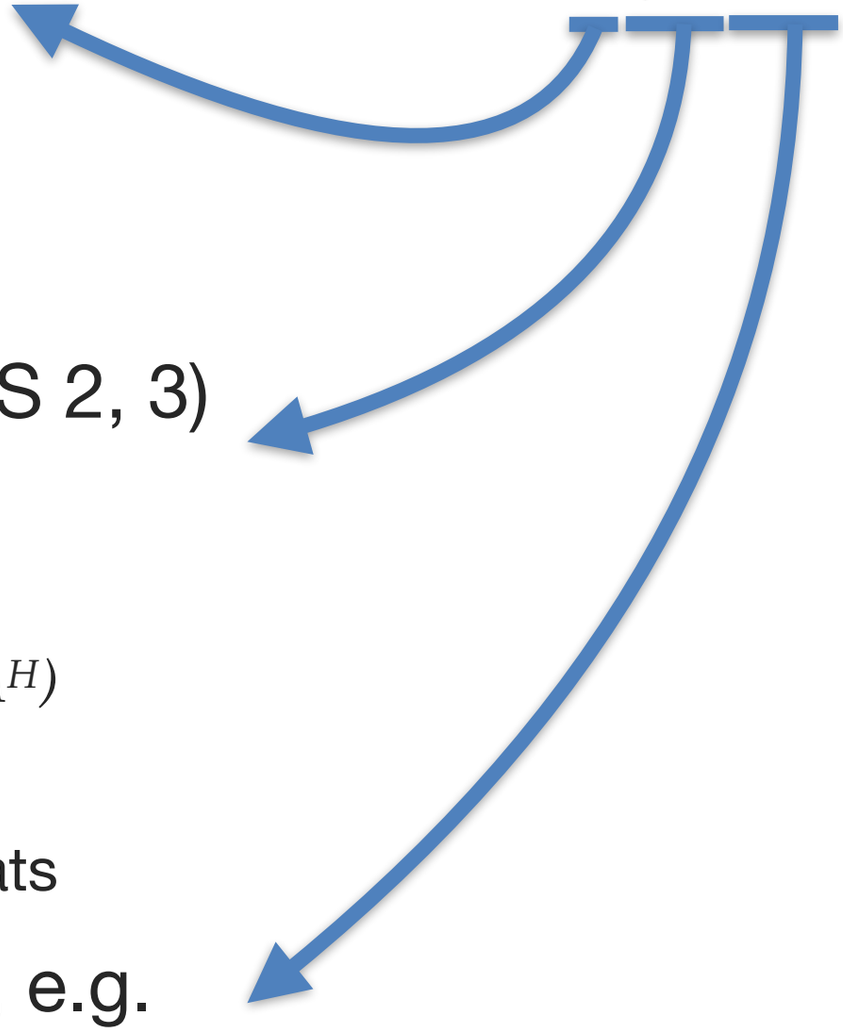
- Stream benchmark
  - https://www.cs.virginia.edu/stream/
  - Add -fopenmp or equivalent to Makefile CFLAGS and FFLAGS
  - Adjust STREAM_ARRAY_SIZE depending on cache size;
    see instructions in code (I changed this to be dynamically allocated)

BLAS naming scheme

- One or two letter **data type** (precision)
  - `i` = integer (e.g., index)
  - `s` = single (float)
  - `d` = double
  - `c` = single-complex
  - `z` = double-complex
- Two letter **matrix type** (BLAS 2, 3)
  - `ge` = general nonsymmetric
  - `sy` = symmetric $(A = A^T)$
  - `he` = complex Hermitian $(A = A^H)$
  - `tr` = triangular (L or U)
  - Also banded and packed formats
- Two or more letter **function**, e.g.
  - `mv` = matrix-vector product
  - `mm` = matrix-matrix product
  - `etc.`

Example

`dgemm` $C = AB + C$



BLAS naming scheme

- One or two letter **data type**
  - `i` = integer (e.g., index)
  - `s` = single (float)
  - `d` = double
  - `c` = single-complex
  - `z` = double-complex
- Two letter **matrix type** (BLAS 2, 3)
  - `ge` = general nonsymmetric
  - `sy` = symmetric $(A = A^T)$
  - `he` = complex Hermitian $(A = A^H)$
  - `tr` = triangular (L or U)
  - Also banded and packed formats
- Two or more letter **function**, e.g.
  - `mv` = matrix-vector product
  - `mm` = matrix-matrix product
  - `etc.`

BLAS 1 examples

`sdot`	$result = x^T y$ (single)
`ddot`	$result = x^T y$ (double)
`cdotc`	$result = x^H y$ (single-complex)
`cdotu`	$result = x^T y$ (single-complex)
`zdotc`	$result = x^H y$ (double-complex)
`zdotu`	$result = x^T y$ (double-complex)

`_axpy`	$y = \alpha\, x + y$
`_scal`	$y = \alpha\, y$
`_copy`	$y = x$
`_swap`	$x \leftrightarrow y$
`_nrm2`	$result = \|x\|_2$
`_asum`	$result = \sum_i \|x_i\|$
`i_amax`	$index = \text{argmax}_i \|x_i\|$
`_rot`	Apply Given's rotation: $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

BLAS naming scheme

- One or two letter **data type**
  - `i` = integer (e.g., index)
  - `s` = single (float)
  - `d` = double
  - `c` = single-complex
  - `z` = double-complex
- Two letter **matrix type** (BLAS 2, 3)
  - `ge` = general nonsymmetric
  - `sy` = symmetric $(A = A^T)$
  - `he` = complex Hermitian $(A = A^H)$
  - `tr` = triangular $(L$ or $U)$
  - Also banded and packed formats
- Two or more letter **function**, e.g.
  - `mv` = matrix-vector product
  - `mm` = matrix-matrix product
  - `etc.`

BLAS 2 examples

`_gemv`	$y = Ax + y,$	A general
`_symv`	$y = Ax + y,$	A symmetric
`_hemv`	$y = Ax + y,$	A Hermitian
`_ger`	$C = xy^T + C,$	C general
`_syr`	$C = xx^T + C,$	C symmetric
`_her`	$C = xx^H + C,$	C Hermitian
`_trmv`	$x = Ax,$	A triangular
`_trsv`	$solve\ Ax = b,$	A triangular

BLAS 3 examples

`_gemm`	$C = AB + C,$	all general
`_symm`	$C = AB + C,$	A symmetric
`_hemm`	$C = AB + C,$	A Hermitian
`_syrk`	$C = AA^T + C,$	C symmetric
`_herk`	$C = AA^H + C,$	C Hermitian
`_trmm`	$X = AX\ or\ XA,$	A triangular
`_trsm`	$solve\ AX = B,$	A triangular

Why is BLAS so important?

- BLAS are efficient, portable, parallel, and widely available
  - Vendors (Intel, AMD, IBM, Cray, NVIDIA, ...) provide highly optimized versions
  - Open source libraries available (ATLAS, OpenBLAS)

- Commonly used for high quality linear algebra and HPC software

- Performance of many applications depends on underlying BLAS

Outline

- Legacy Software
  - BLAS
  - **LINPACK**
  - LAPACK
  - ScaLAPACK

- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

LINPACK

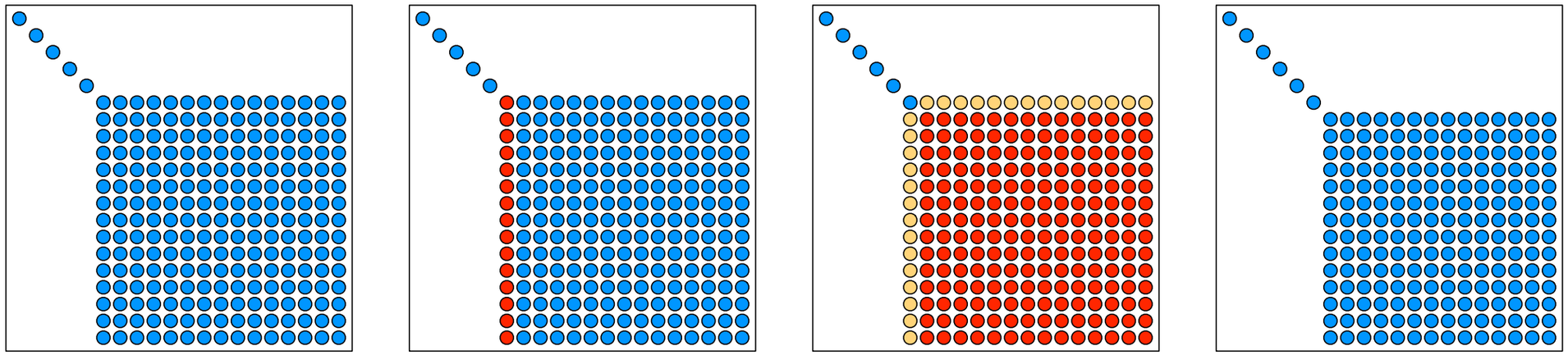
LINear algebra PACKage

- Math software library
  - solving dense & banded linear systems
  - singular value decomposition (SVD)
- Written in 1970s using Fortran 66
- Aiming for software portability and efficiency
- Level 1 BLAS
- Four primary contributors:
  - **Jack Dongarra**, Argonne
  - **Jim Bunch**, U. California, San Diego
  - **Cleve Moler**, New Mexico
  - **Pete Stewart**, U. of Maryland
- Superseded by LAPACK

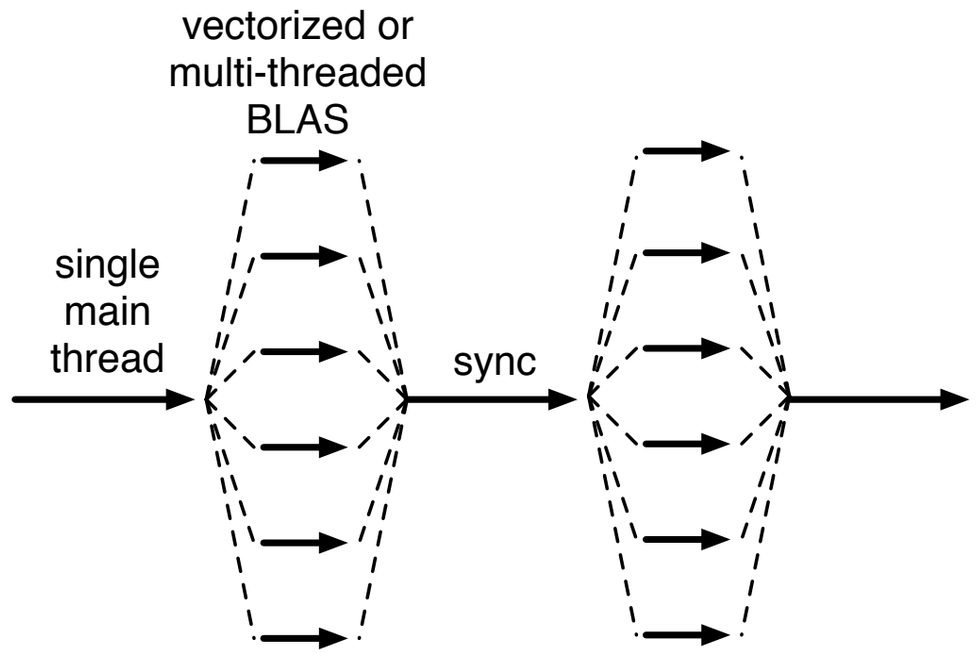
Computing in 1974

- High performance computers
  - IBM 370/195
  - CDC 7600
  - Univac 1110
  - DEC PDP-10
  - Honeywell 6030
- Fortran 66
- EISPACK released in 1974
  - Eigenvalue and singular value problems
  - Translation of Algol into Fortran
  - Did not use BLAS
- Level 1 BLAS — vector operations (1979)
  - Aimed at vector supercomputer architectures
- LINPACK released in 1979
  - About time of Cray 1

LU Factorization in LINPACK



- Factor one column at a time
  - i_amax and _scal
- Update each column of trailing matrix, one column at a time
  - _axpy
- Level 1 BLAS
- Bulk synchronous
  - Single main thread
  - Parallel work in BLAS
  - “Fork-and-join” model



Outline

- Legacy Software
  - BLAS
  - LINPACK
  - **LAPACK**
  - ScaLAPACK

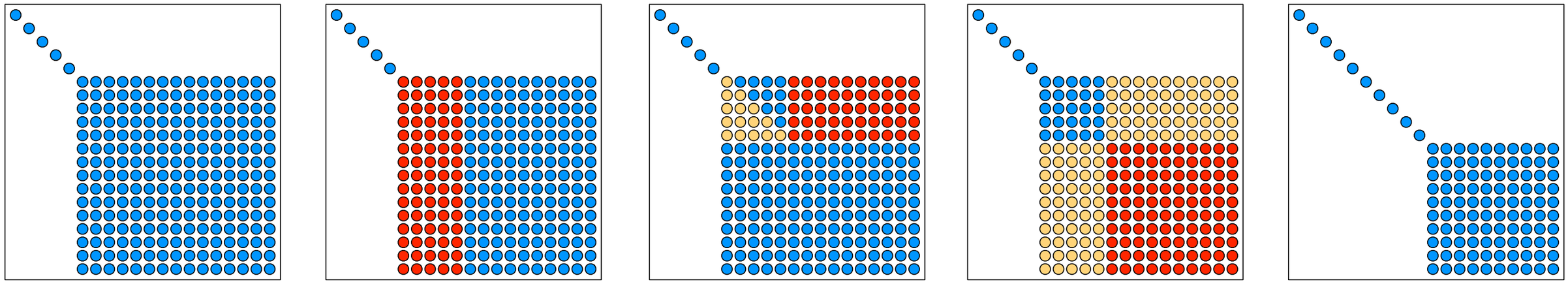
- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

LAPACK

Linear Algebra PACKage

- Reformulate linear algebra algorithms as **blocked algorithms** using BLAS-3
- Linear systems
  - Nearly 100% BLAS-3
- Singular value and symmetric eigenvalue problems
  - About 50% BLAS-3
- Nonsymmetric eigenvalue problems
  - About 80% BLAS-3
- 4 data types: single, double, single-complex, double-complex

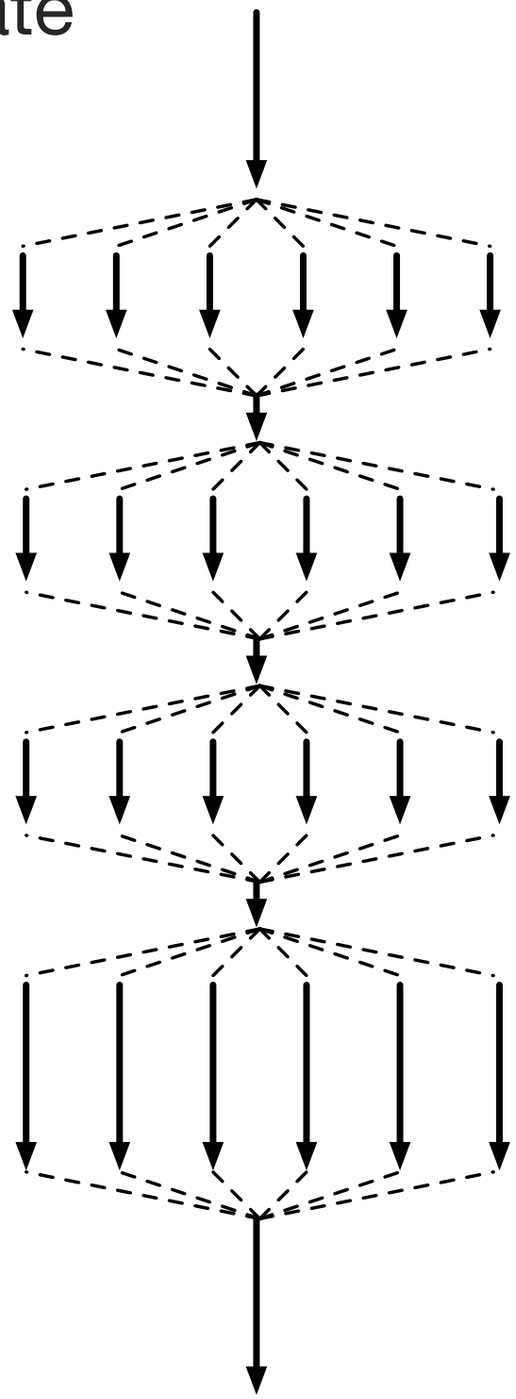
LU factorization in LAPACK

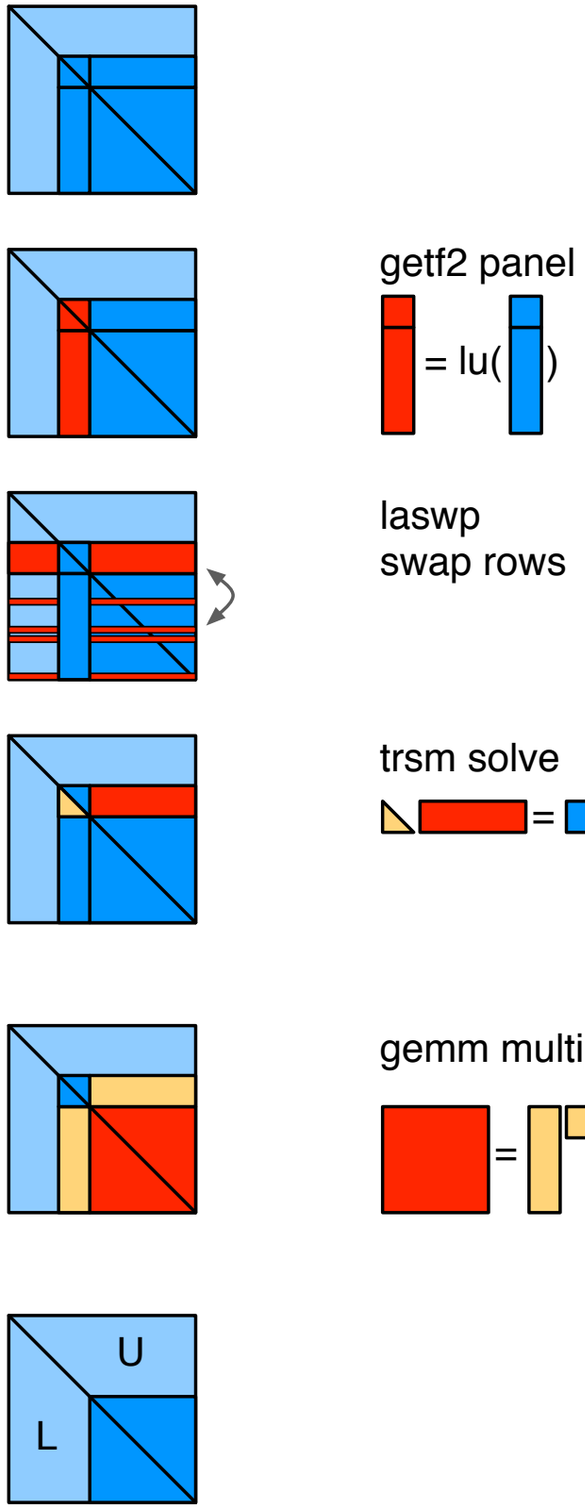


- Factor panel of n_b columns
  - getf2, unblocked BLAS-2 code
- Level 3 BLAS update block-row of U
  - trsm
- Level 3 BLAS update trailing matrix
  - gemm
  - Aimed at machines with cache hierarchy
- Bulk synchronous

Parallelism in LAPACK

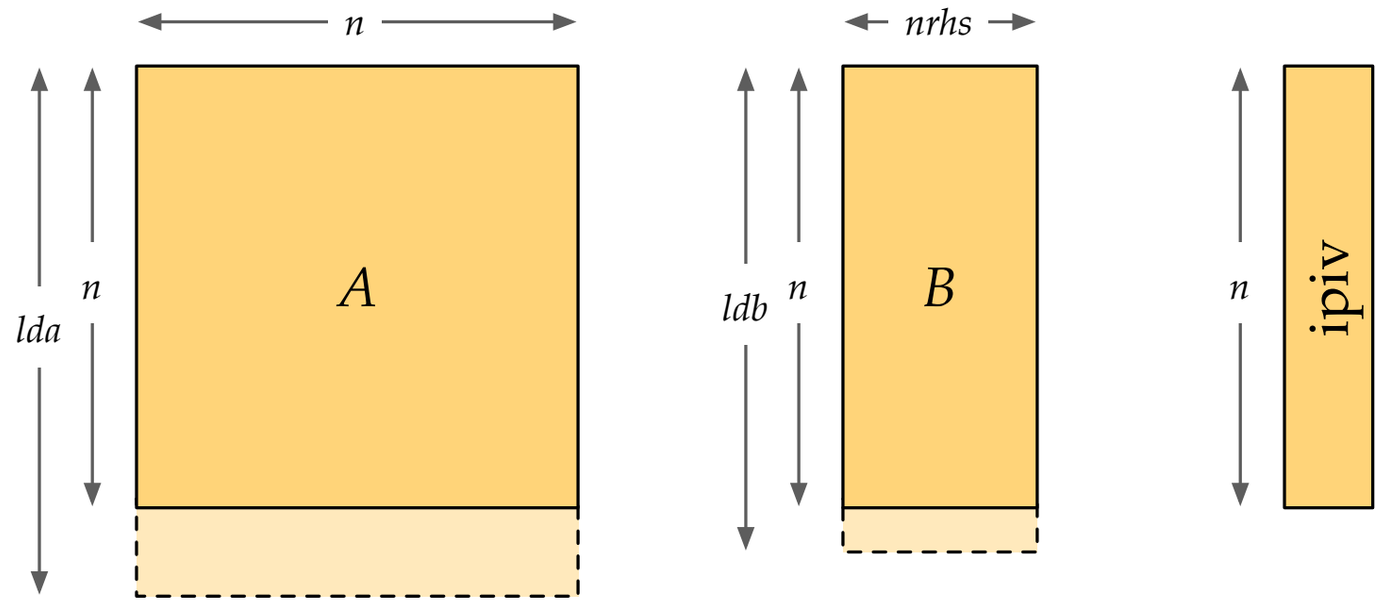
- Most flops in gemm update
  - 2/3 n^3 term
  - Easily parallelized using multi-threaded BLAS
  - Done in any reasonable software
- Other operations lower order
  - Potentially expensive if not parallelized



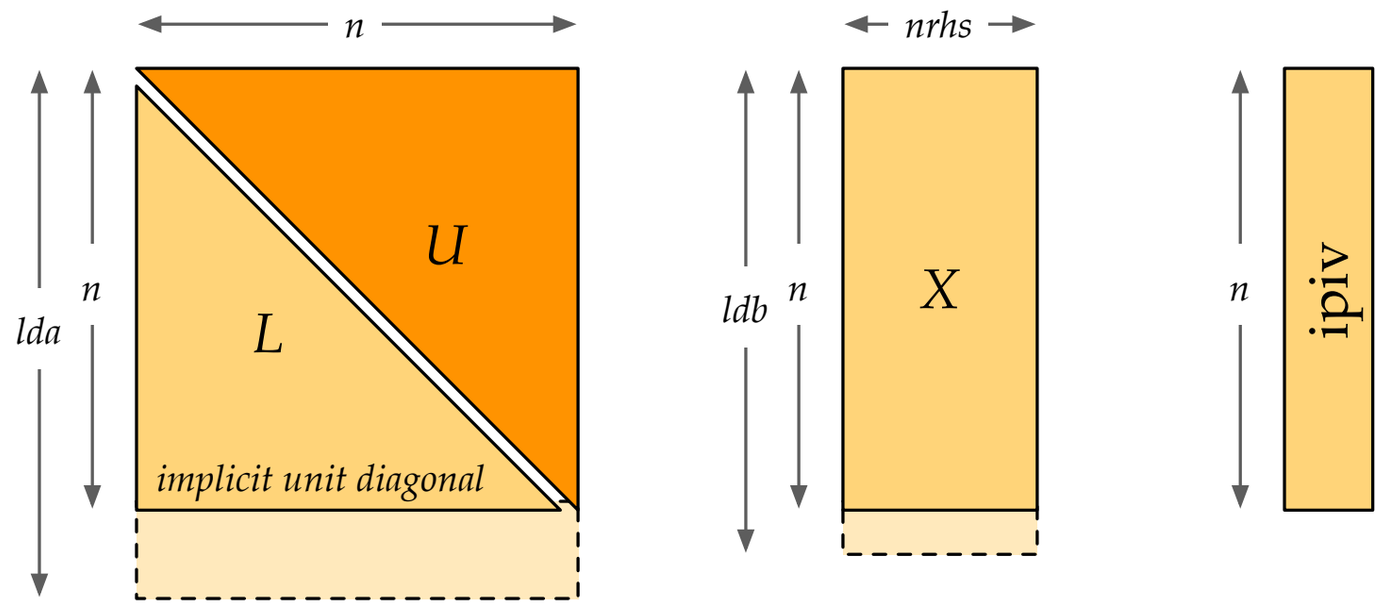


LAPACK routine, solve $AX = B$

- `dgesv( n, nrhs, A, lda, ipiv, B, ldb, info )`
- input:



- output:



info (error code)
- = 0: no error
- < 0: invalid argument
- > 0: numerical error (e.g., singular)

L, U overwrite A

X overwrites B

Matrices stored column-wise

LAPACK naming

- Similar to BLAS naming
  - Data type + matrix type + function

- Additional matrix types:
  - po = real symmetric / complex Hermitian **positive definite** (SPD / HPD, all $\lambda > 0$)
  - or = **orthogonal** $(A^T A = I)$
  - un = **unitary** $(A^H A = I)$
  - Also banded, packed, etc. formats

LAPACK routines

- ~480 routines x 4 data types (s, d, c, z)
- **Driver routines**: solve an entire problem
  - `sv` Solve linear system: $Ax = b$
    - `gesv` general non-symmetric (LU)
    - `posv` symmetric positive definite (Cholesky)
    - `sysv` symmetric indefinite (LDL^T)
    - Also packed, banded, tridiagonal storage
  - ls Linear least squares: $Ax \cong b$, A is rectangular
    - `gglse` linear equality-constrained least squares
    - `ggglm` general Gauss-Markov linear model
  - `ev` Eigenvalue decomposition: $Ax = \lambda x$ and $Ax = \lambda \mathrm{M} x$
    - `syevd` symmetric and symmetric generalized
    - `geev` non-symmetric and non-symmetric generalized
  - `svd` Singular value decomposition: $A = U\Sigma V^H$
    - `gesvd` standard and generalized
    - `gesdd` D&C (faster)

http://www.icl.utk.edu/~mgates3/docs/
http://www.icl.utk.edu/~mgates3/docs/lapack.html

LAPACK routines

- **Computation routines**: one step of problem
  - `trf` triangular factorization (LU, Cholesky, LDL^T)
  - `trs` triangular solve
  - `qrf` orthogonal QR factorization
  - `mqr` multiply by Q
  - `gqr` generate Q
  - etc.
- **Auxiliary routines**, begin with “la”
  - `lan__` matrix norm (one, inf, Frobenius, max); see SVD for 2-norm
  - `lascl` scale matrix
  - `lacpy` copy matrix
  - `laset` set matrix & diagonal to constants
  - etc.

http://www.icl.utk.edu/~mgates3/docs/
http://www.icl.utk.edu/~mgates3/docs/lapack.html

Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - **ScaLAPACK**

- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

ScaLAPACK

Scalable Linear Algebra PACKage

- Distributed memory

- Message Passing
  - Clusters of SMPs
  - Supercomputers

- Dense linear algebra

- Modules
  - PBLAS: Parallel BLAS
  - BLACS: Basic Linear Algebra Communication Subprograms

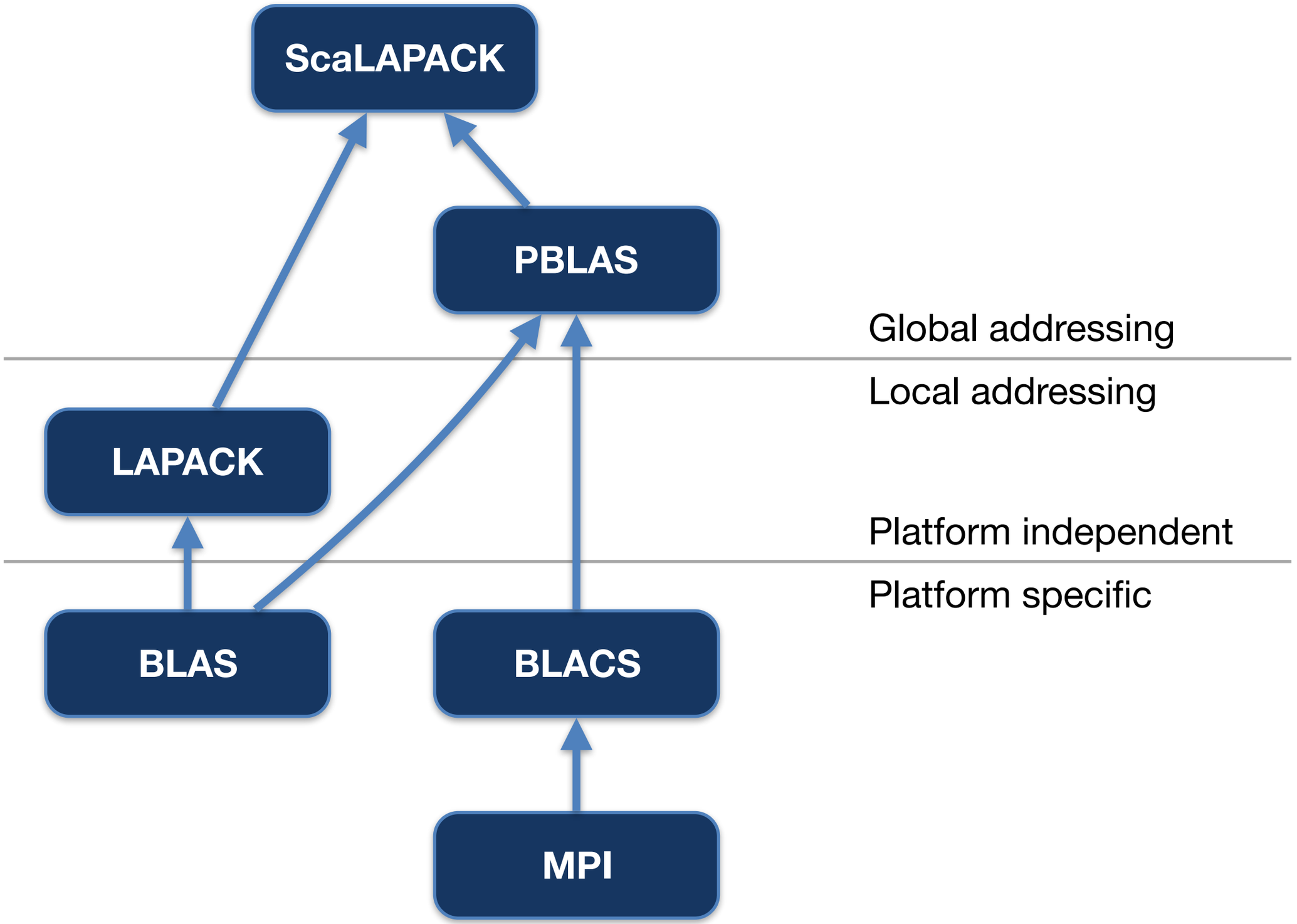
PBLAS

- Similar to BLAS in functionality and naming
- Built on BLAS and BLACS
- Provide global view of matrix

- LAPACK: `dge___( m, n, A(ia, ja), lda, ... )`
  - Submatrix offsets implicit in pointer

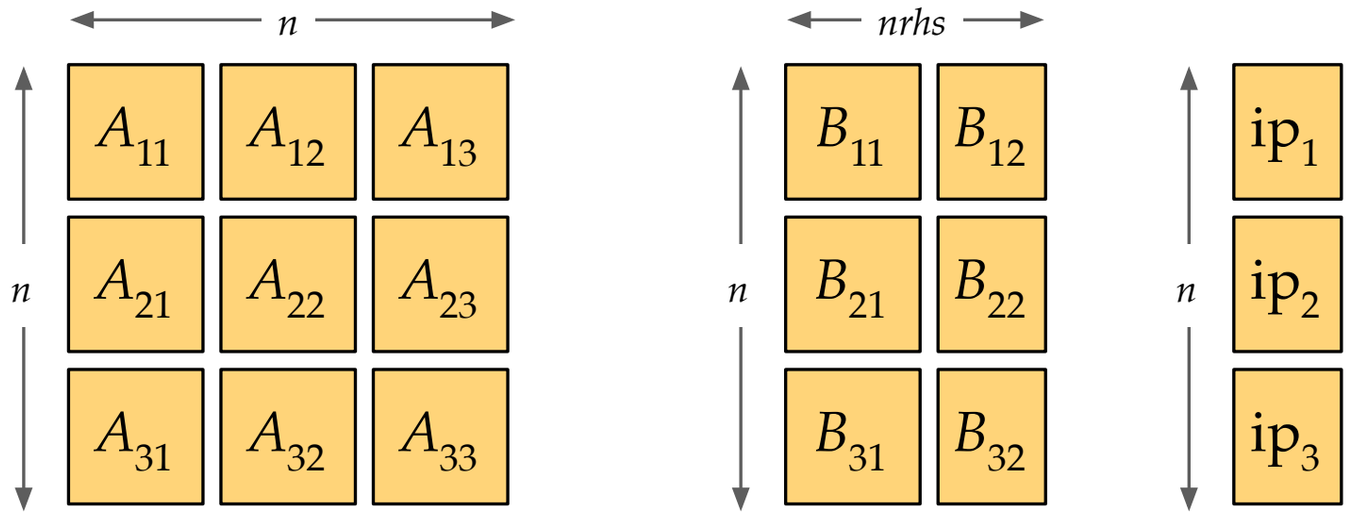
- ScaLAPACK: `pdge___( m, n, A, ia, ja, descA, ... )`
  - Pass submatrix offsets and matrix descriptor

ScaLAPACK structure



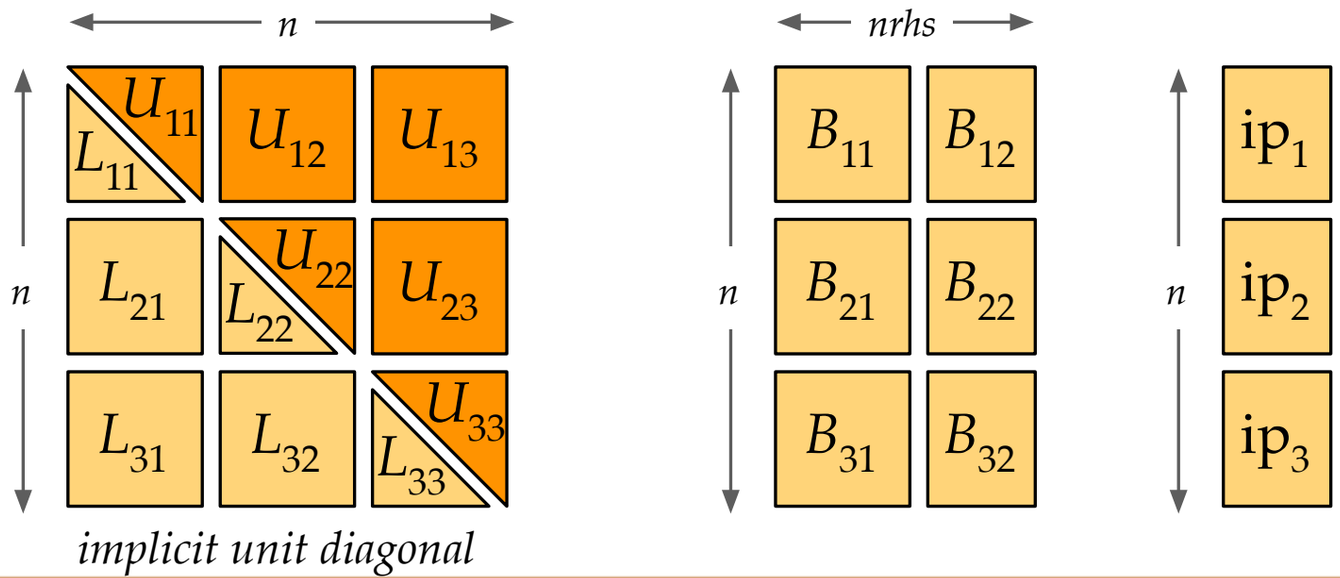
ScaLAPACK routine, solve $AX = B$

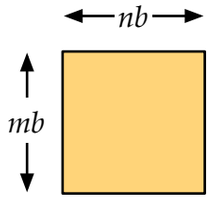
- LAPACK: `dgesv(n, nrhs, A, lda, ipiv, B, ldb, info)`
- ScaLAPACK: `pdgesv(n, nrhs, A, ia, ja, descA, ipiv, B, ib, jb, descB, info)`
- input:



Global matrix point of view

- output:





implicit unit diagonal

info (error code)
- = 0: no error
- < 0: invalid argument
- > 0: numerical error (e.g., singular)

L, U overwrite A

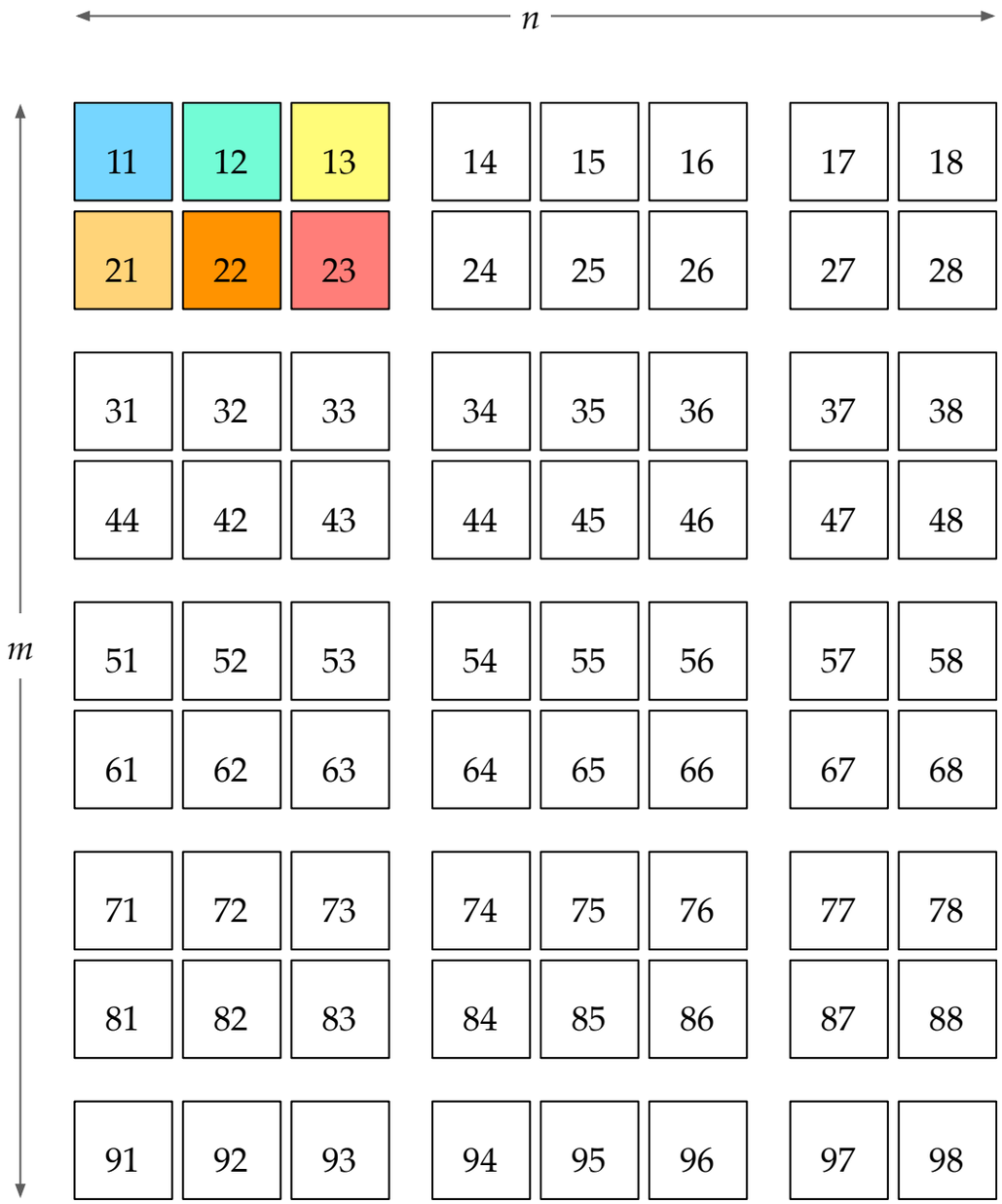
X overwrites B

2D block-cyclic layout

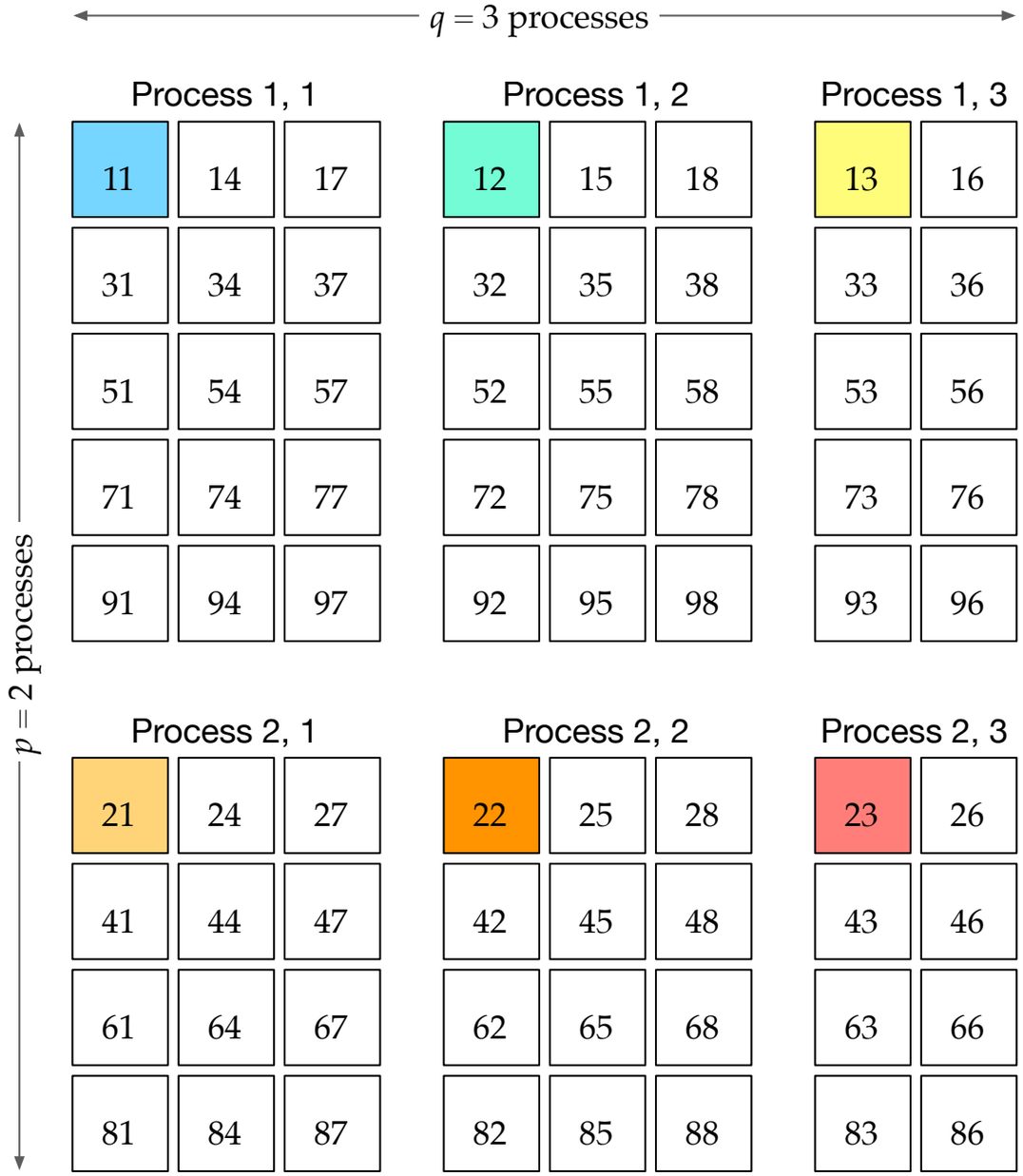
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view



ICL

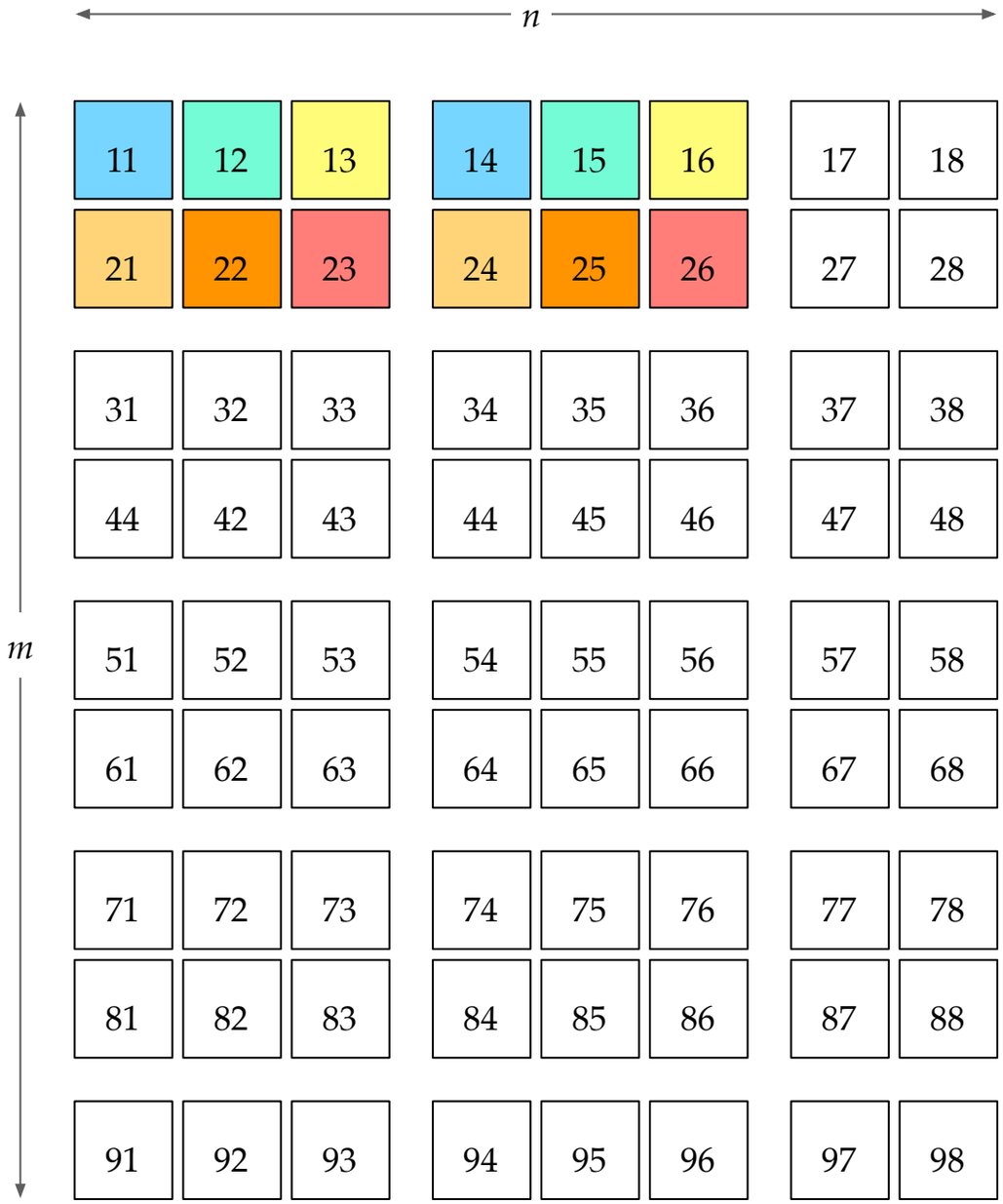
THE UNIVERSITY OF TENNESSEE KNOXVILLE

2D block-cyclic layout

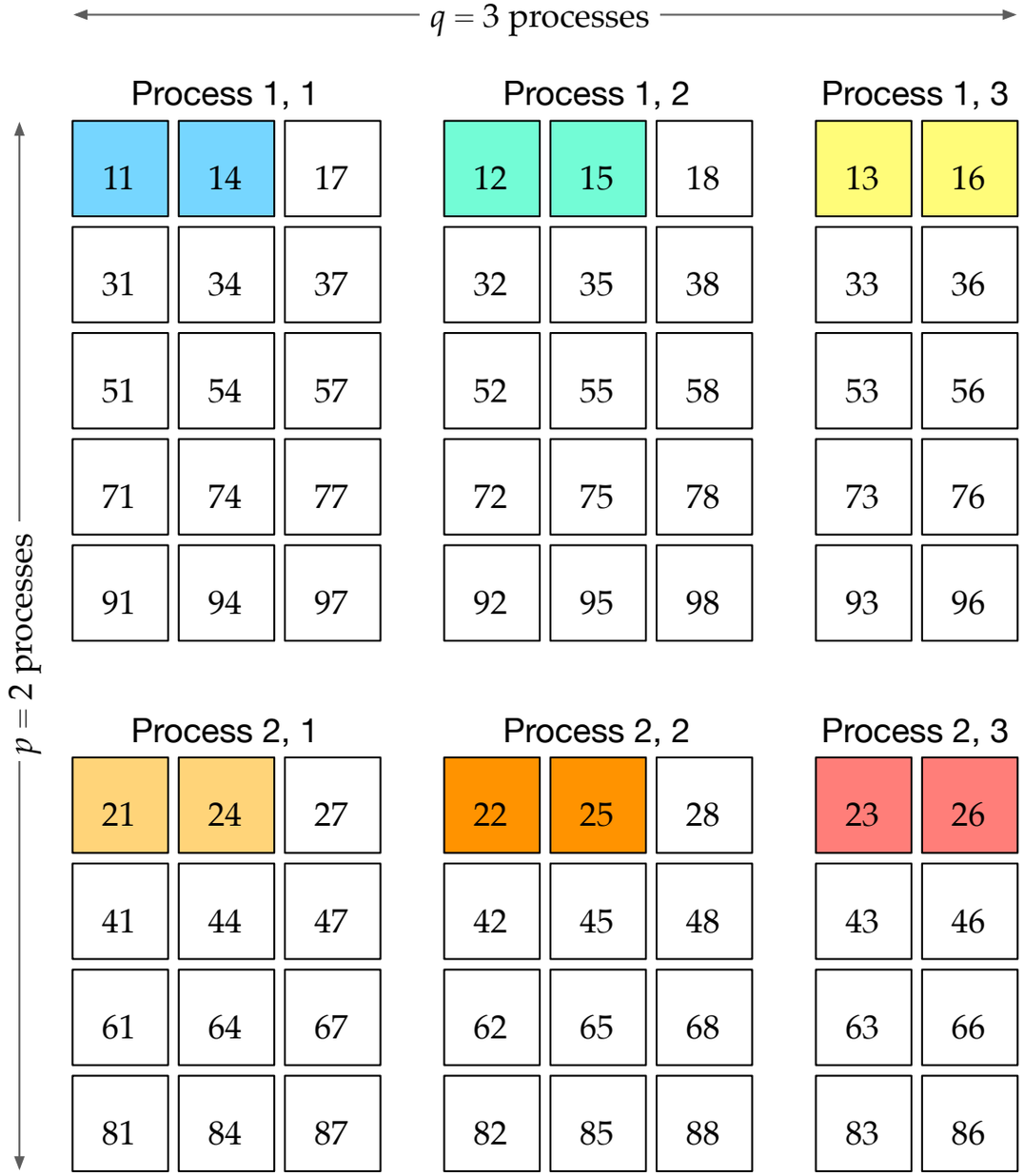
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view



ICL

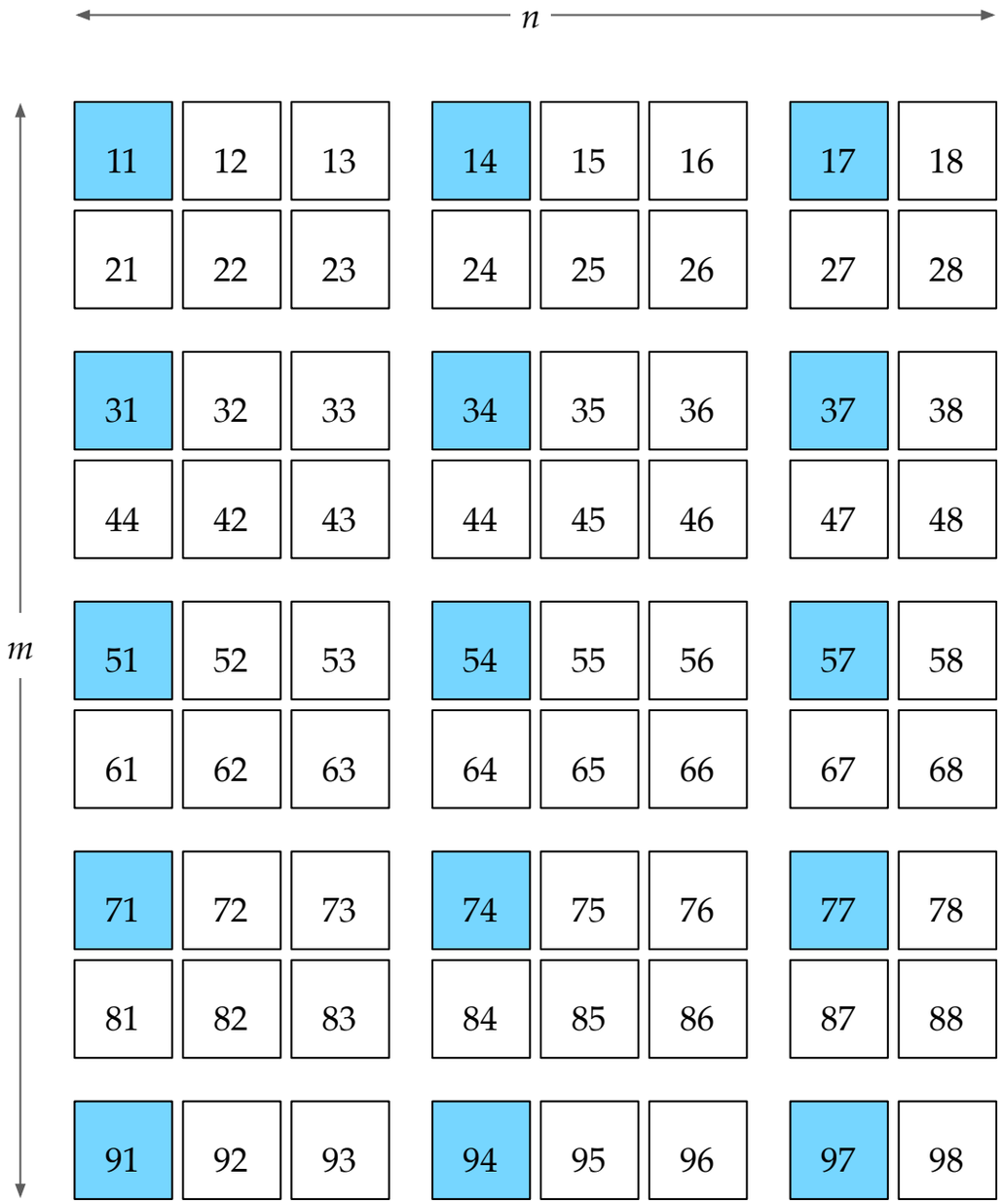
THE UNIVERSITY OF TENNESSEE KNOXVILLE

2D block-cyclic layout

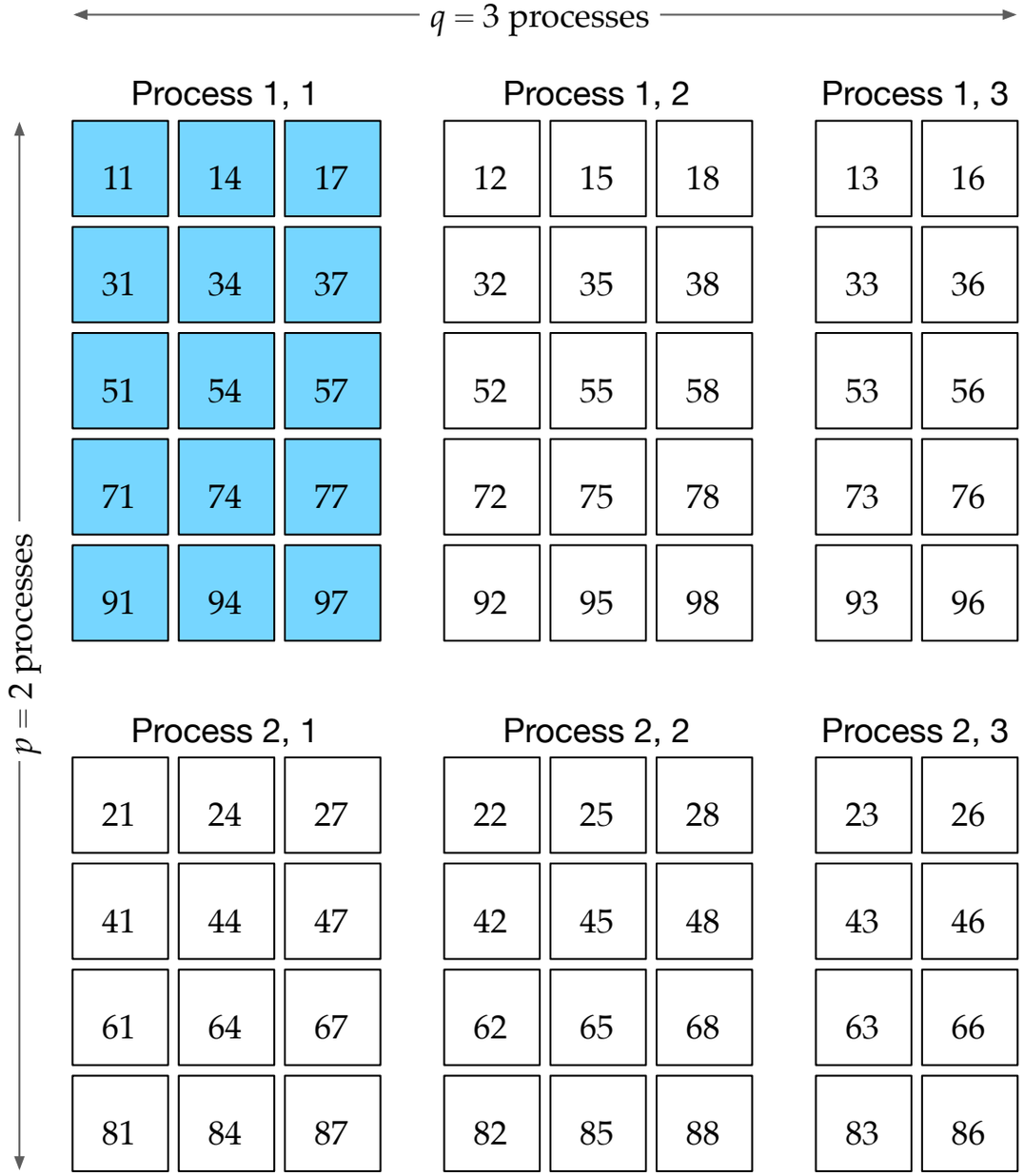
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view



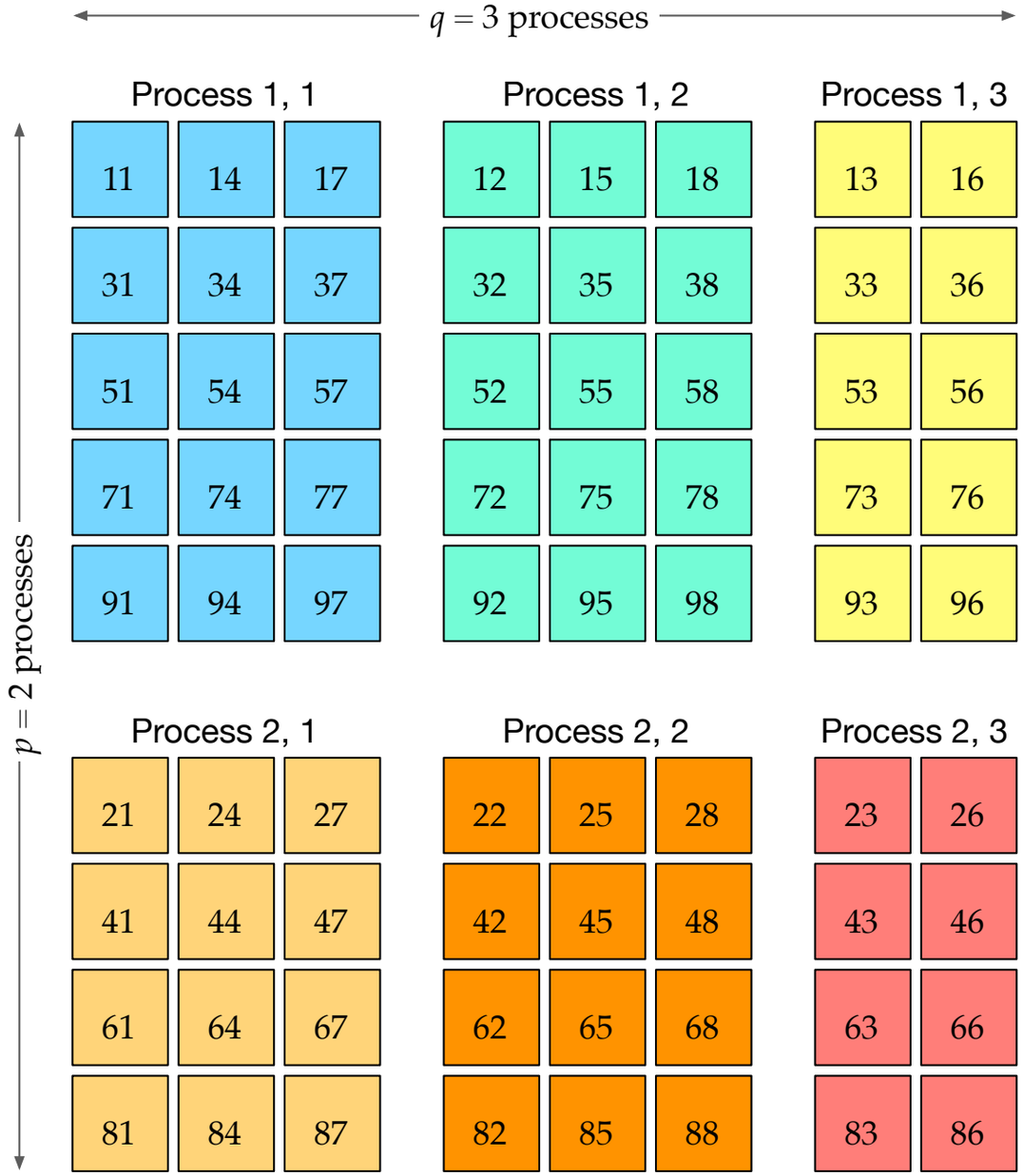
2D block-cyclic layout

$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view

Local process point of view

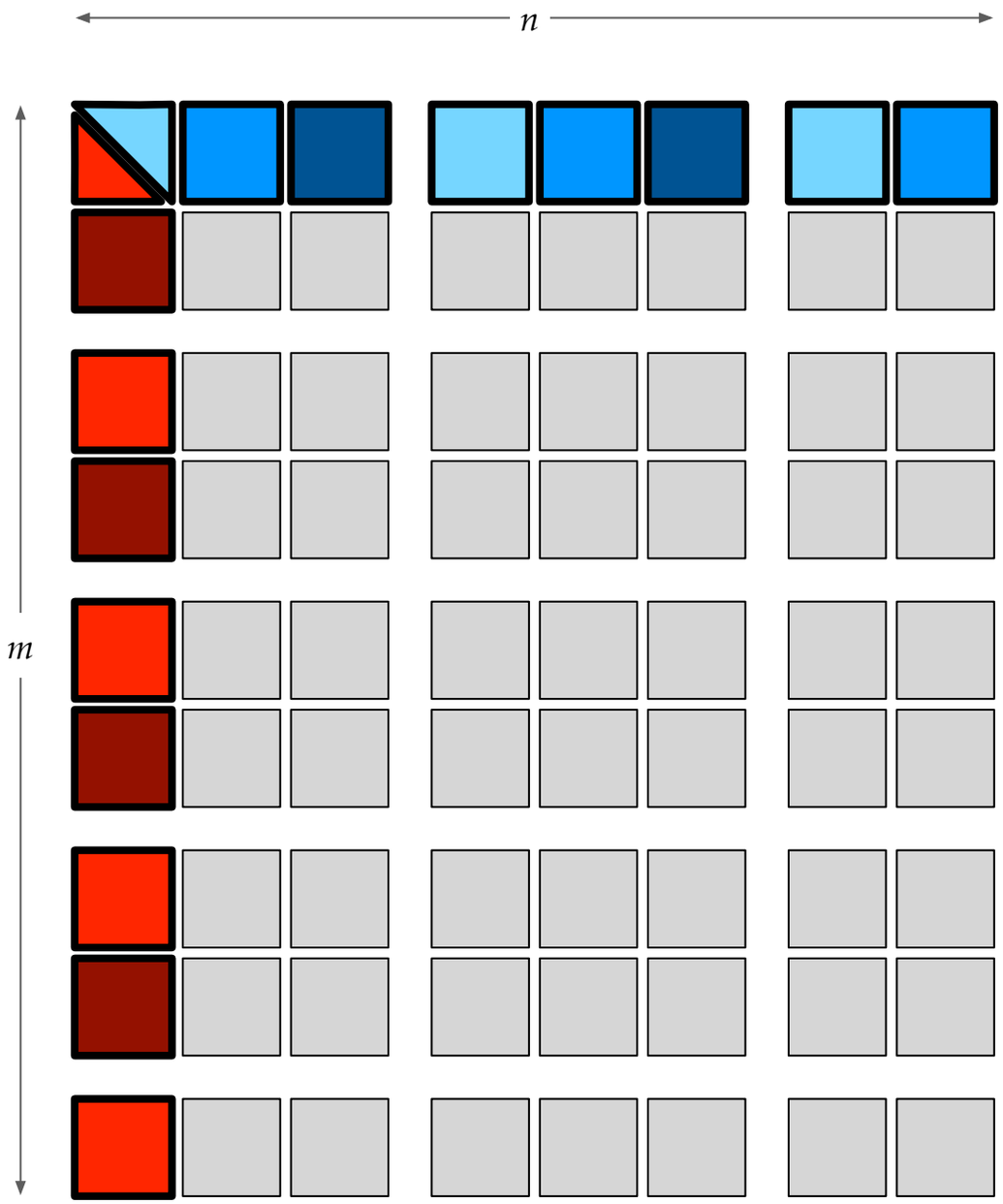


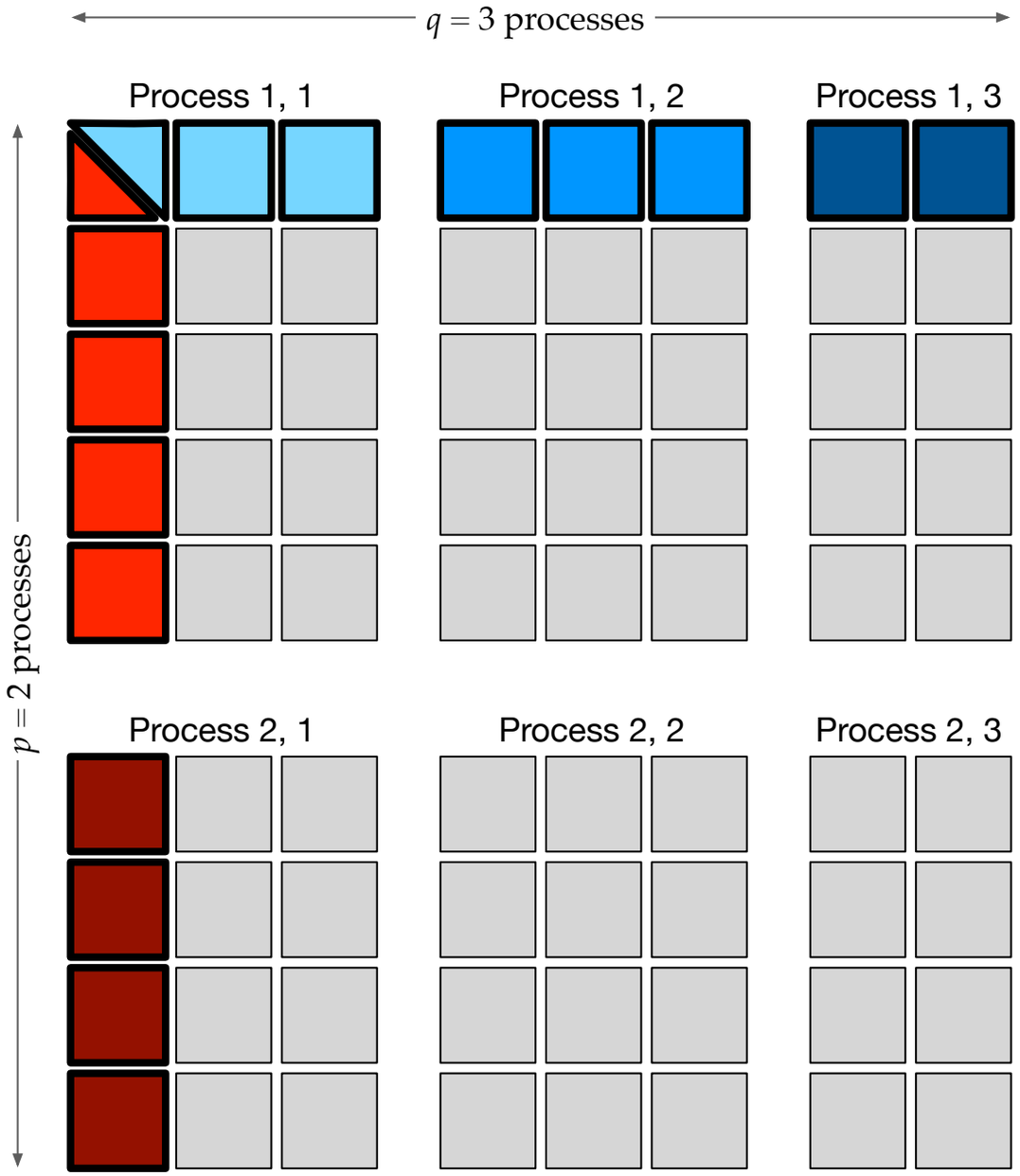
ICL

THE UNIVERSITY OF TENNESSEE KNOXVILLE

LU factorization

- Bold is LU panel (reds) and triangular solve (blues)
- Gray is trailing matrix update



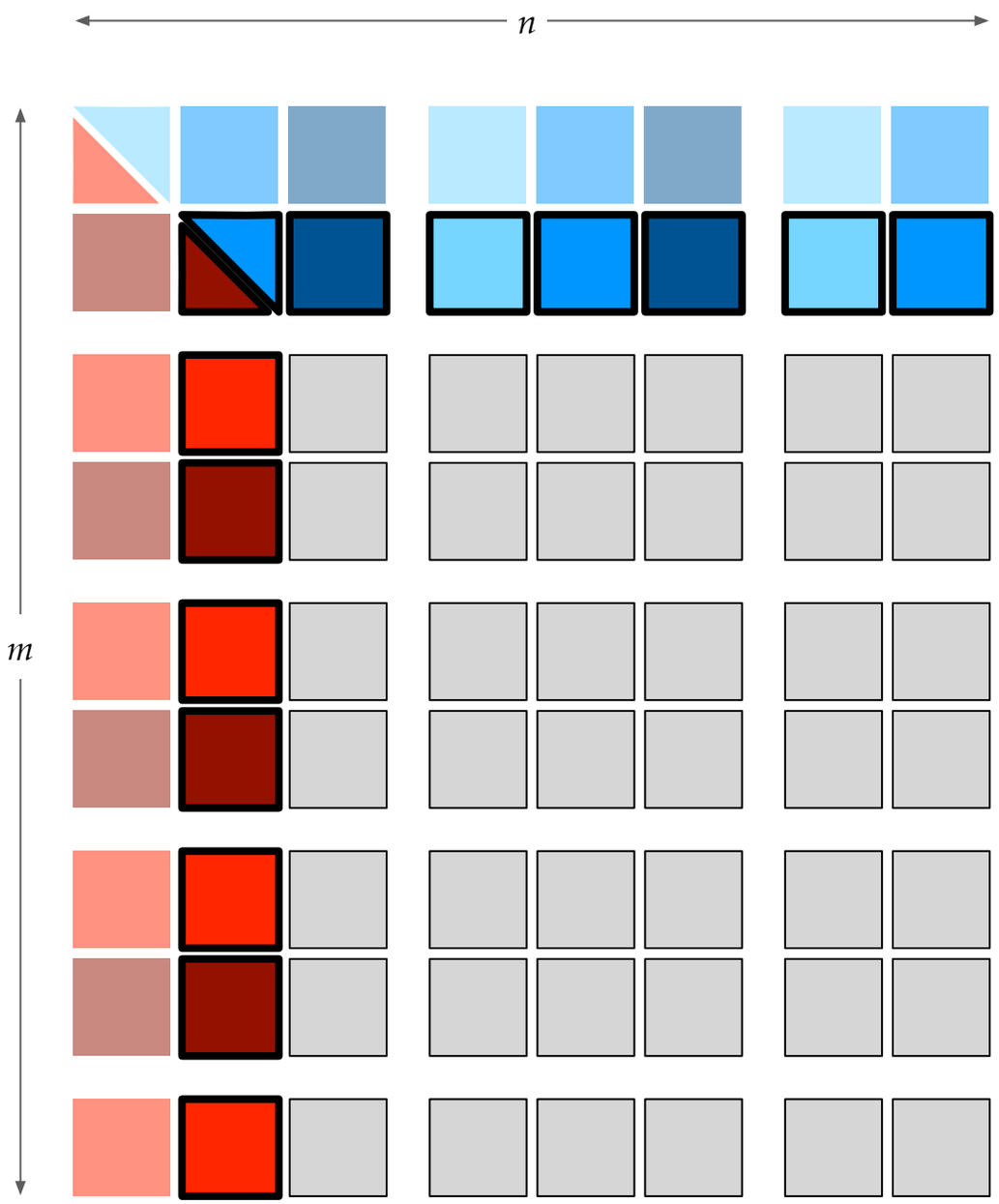


LU factorization

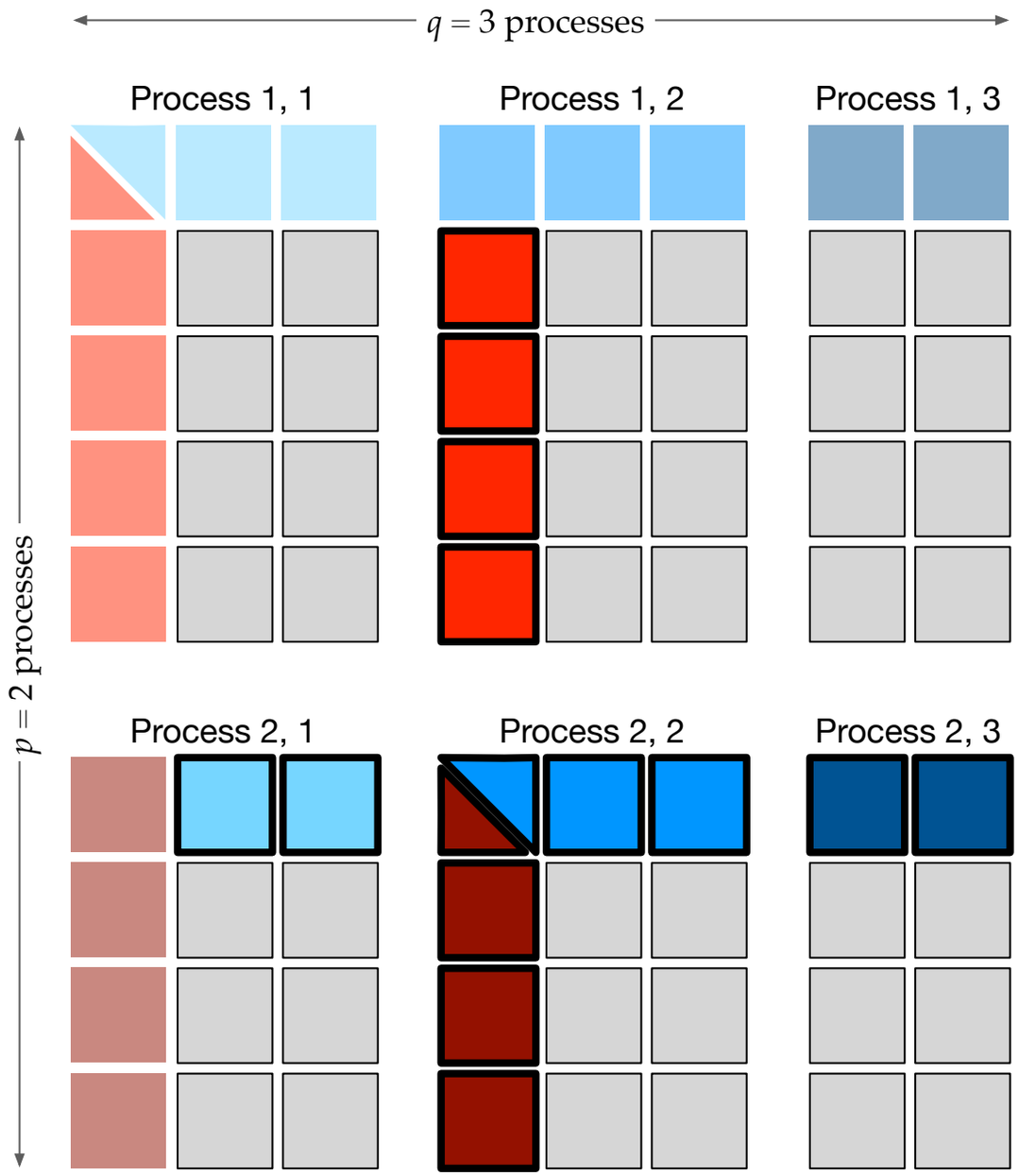
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view



ICL

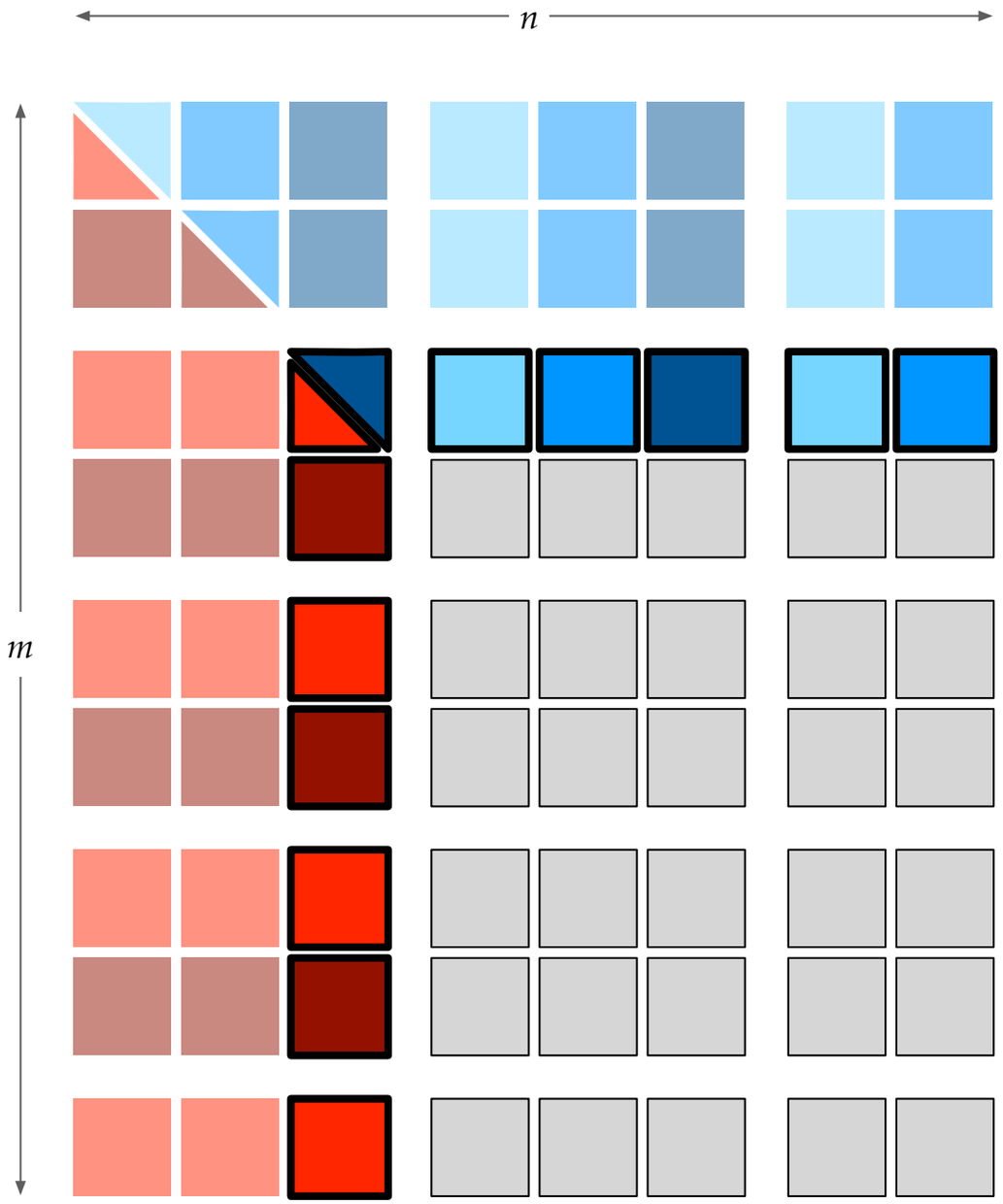
THE UNIVERSITY OF TENNESSEE KNOXVILLE

LU factorization

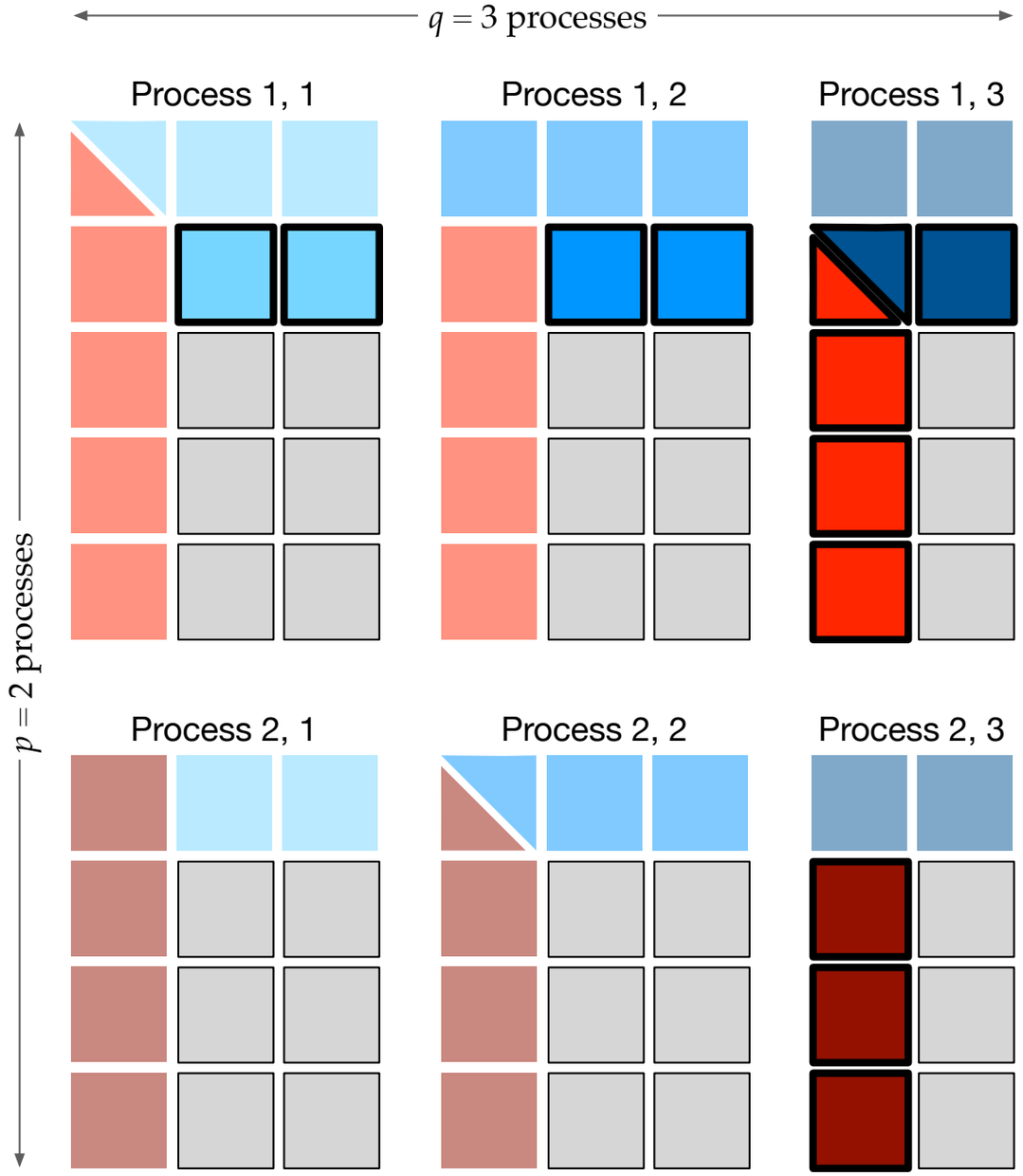
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

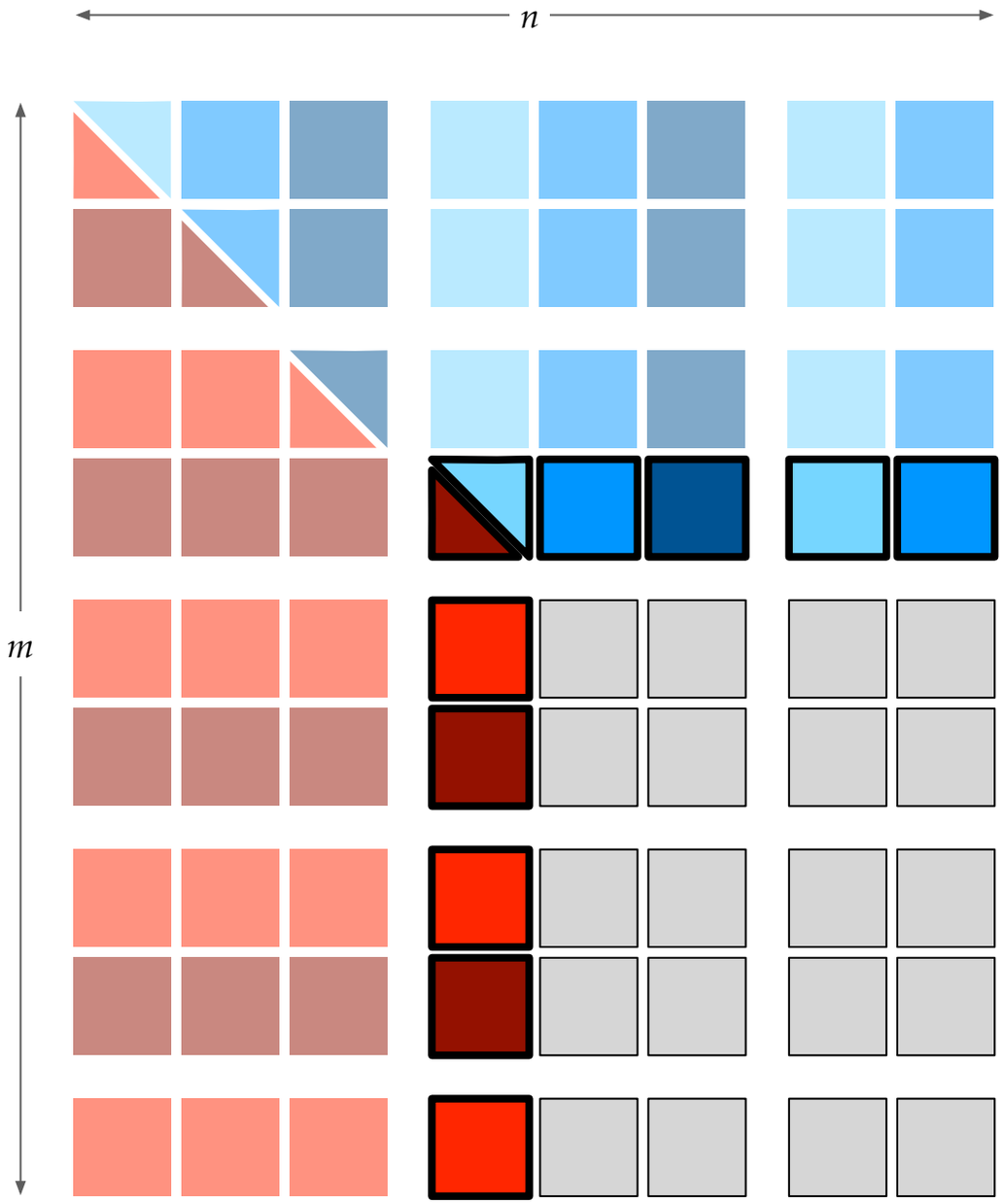


LU factorization

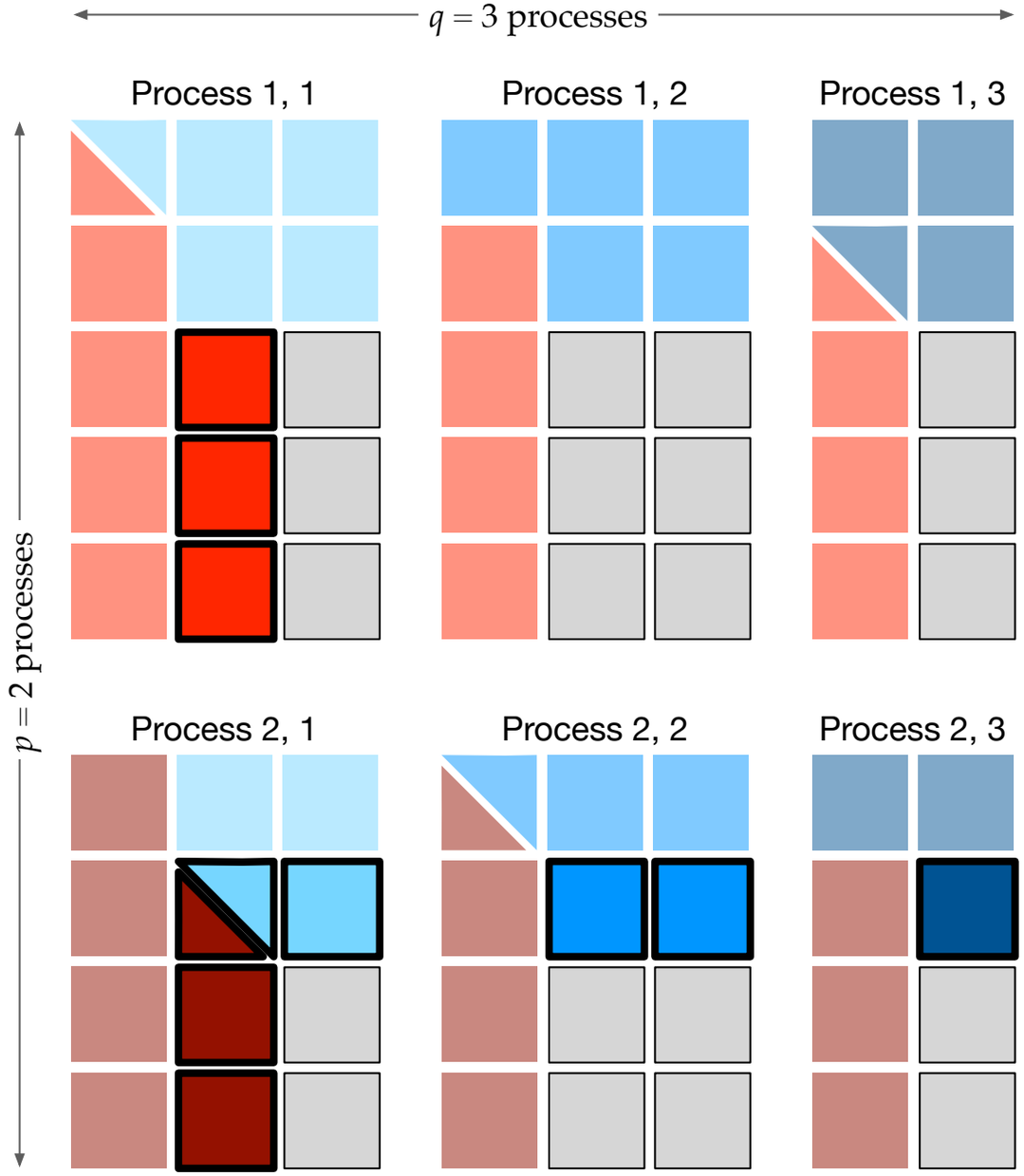
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

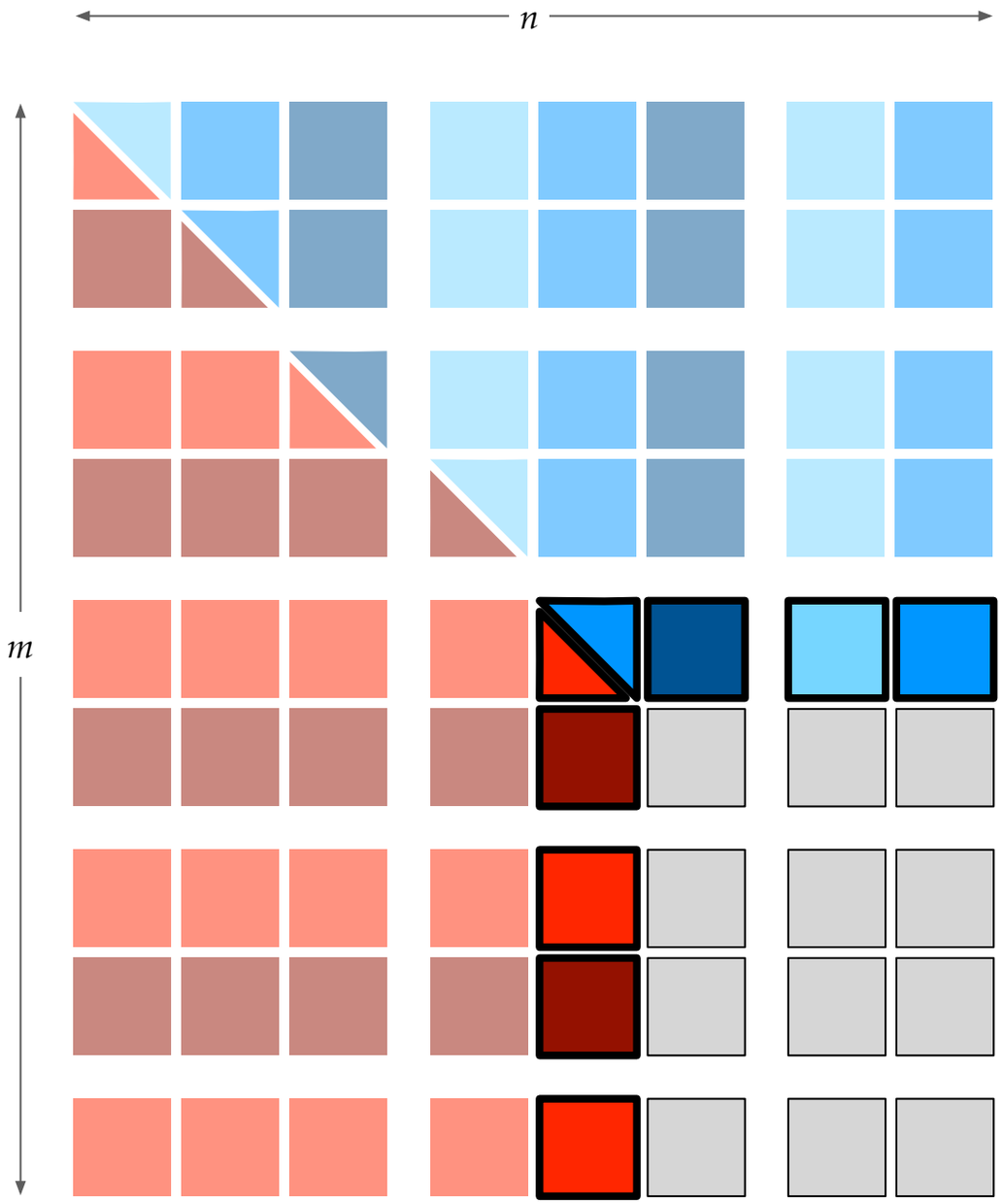


LU factorization

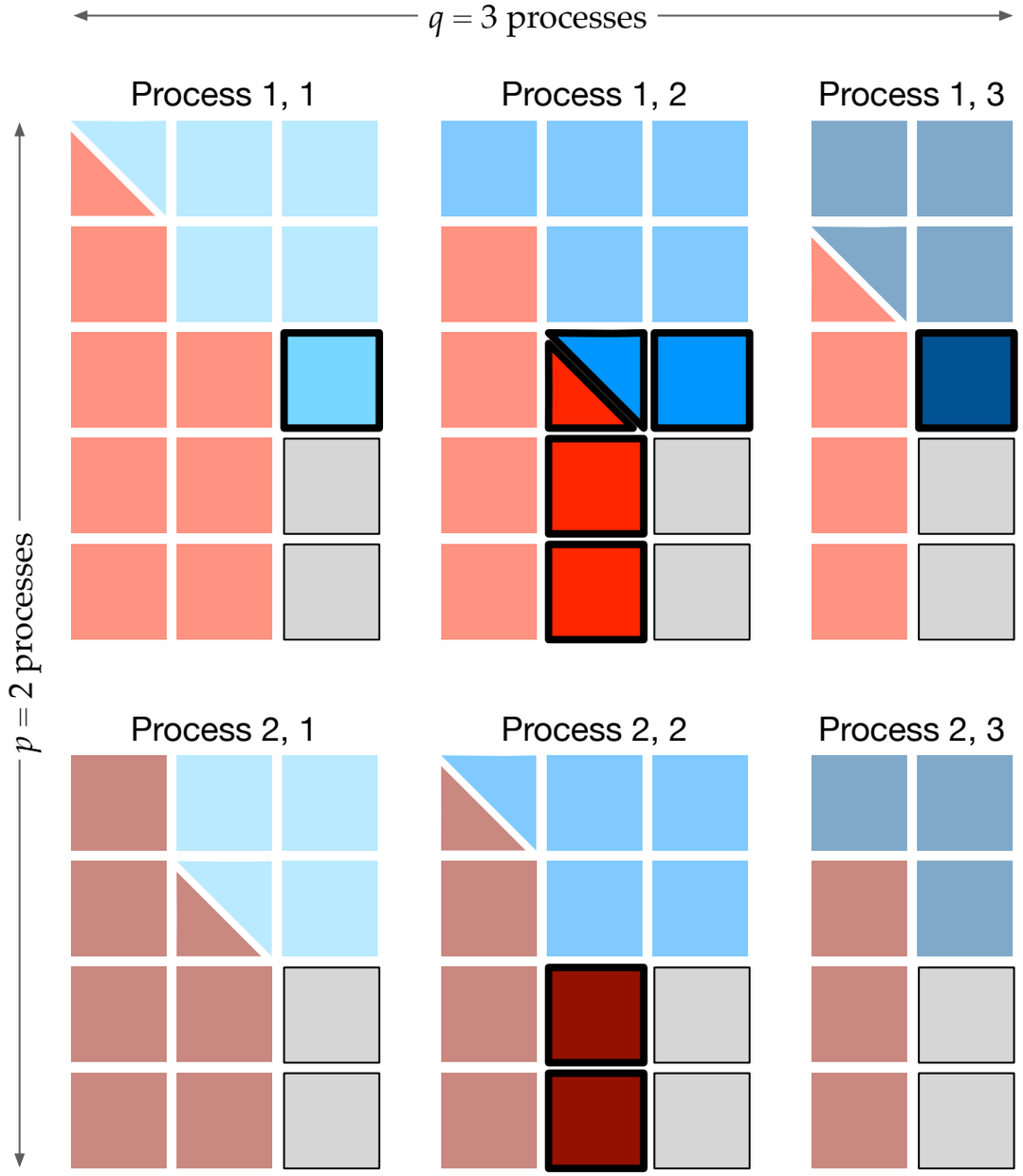
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

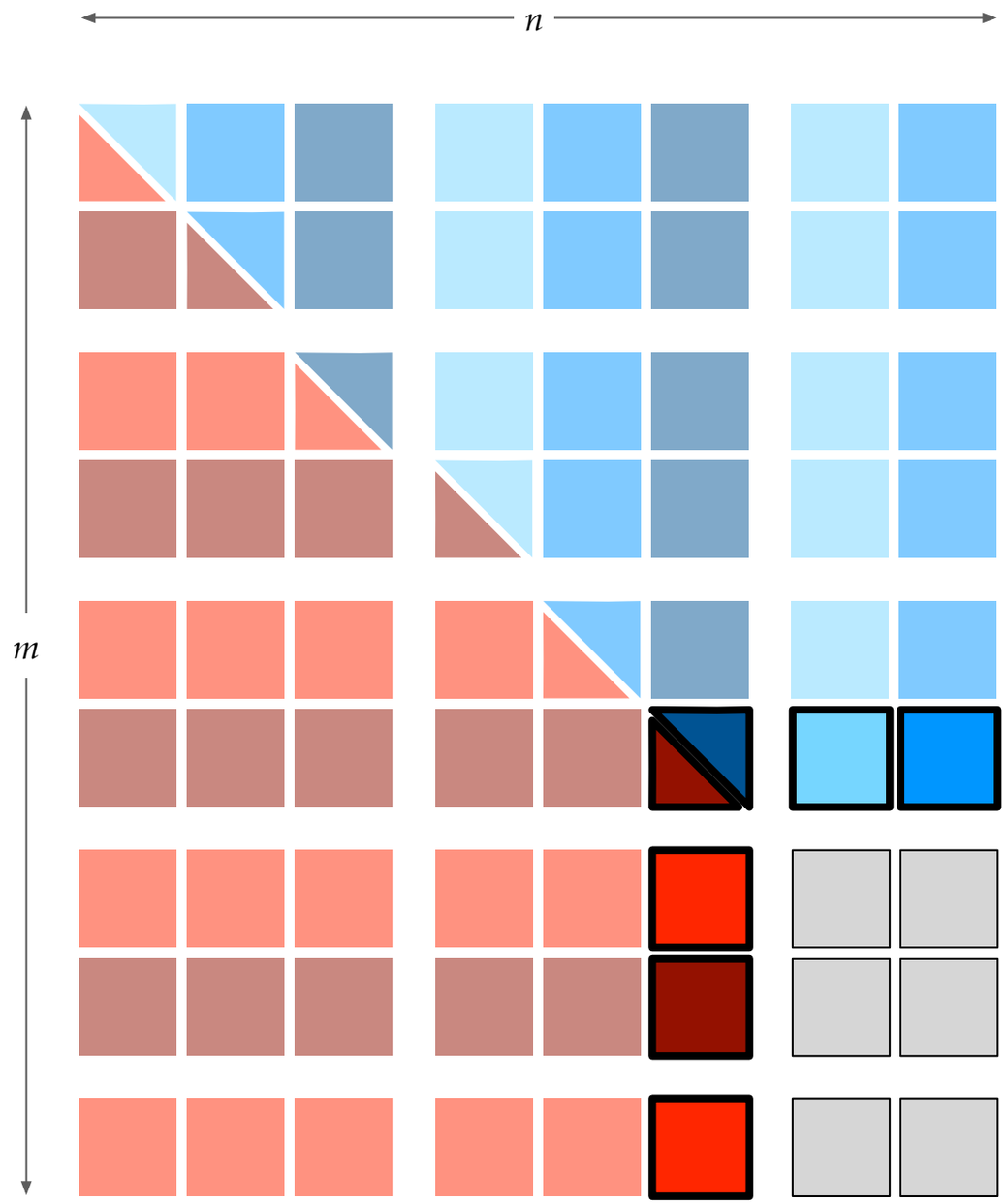


LU factorization

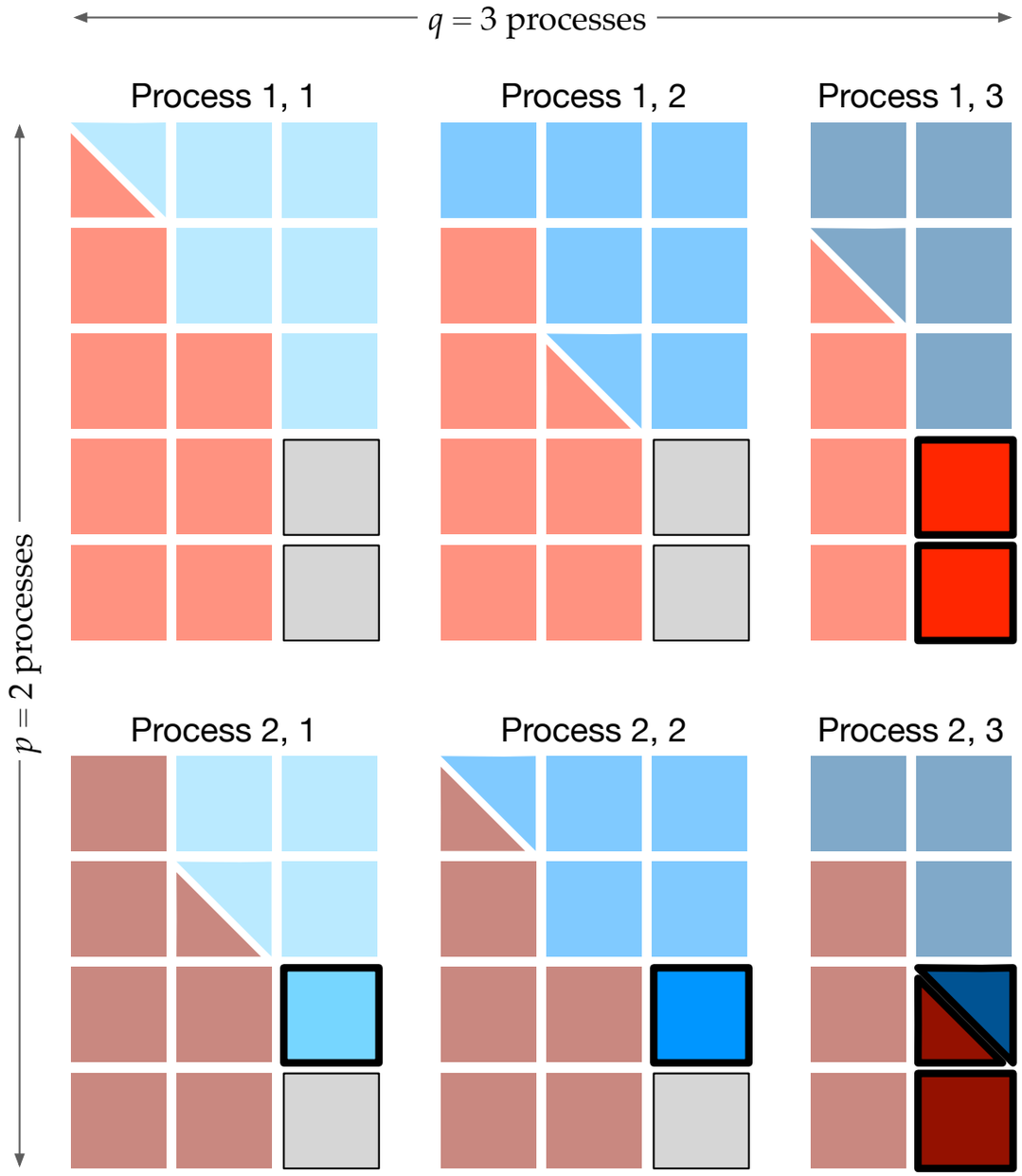
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

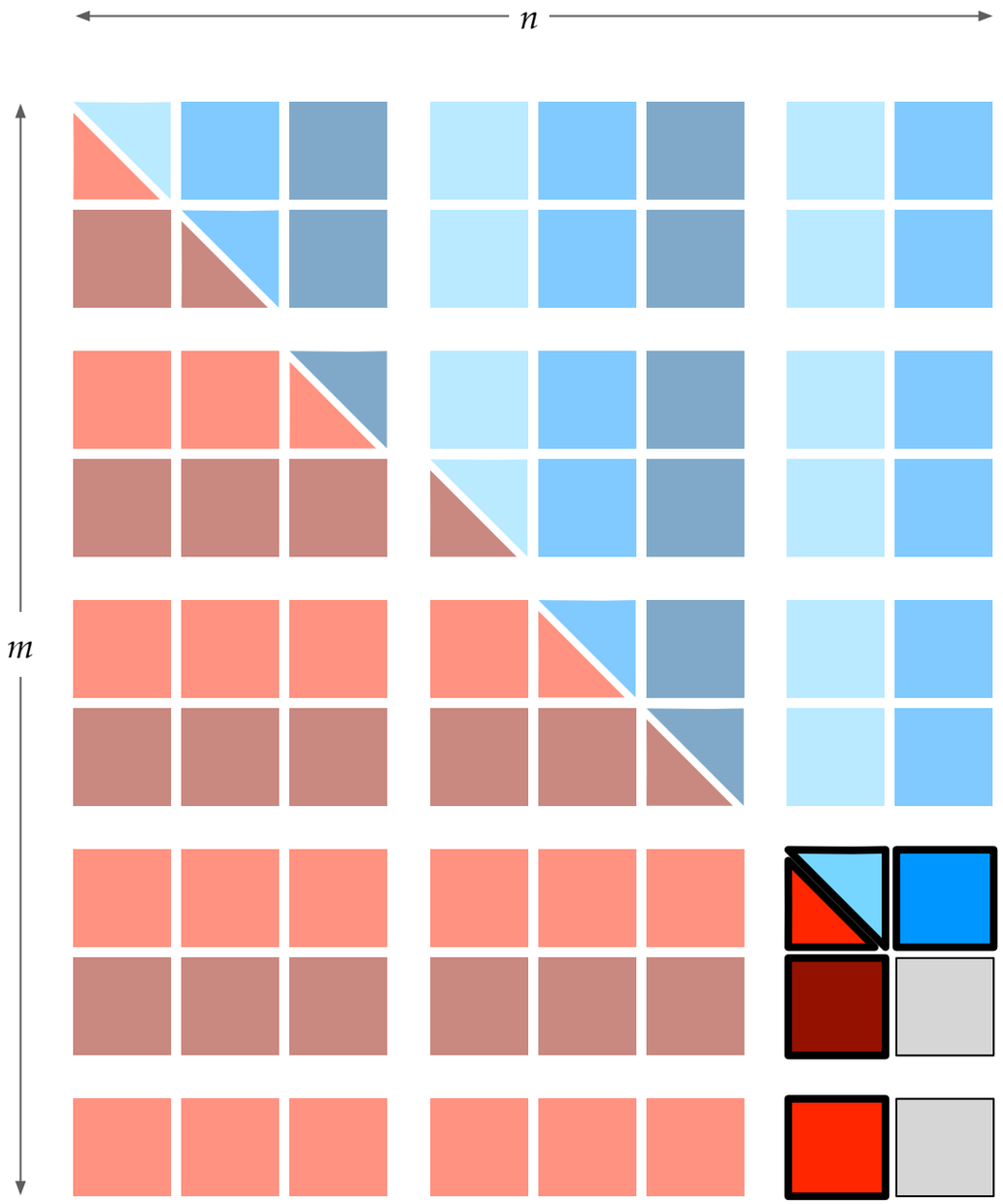


LU factorization

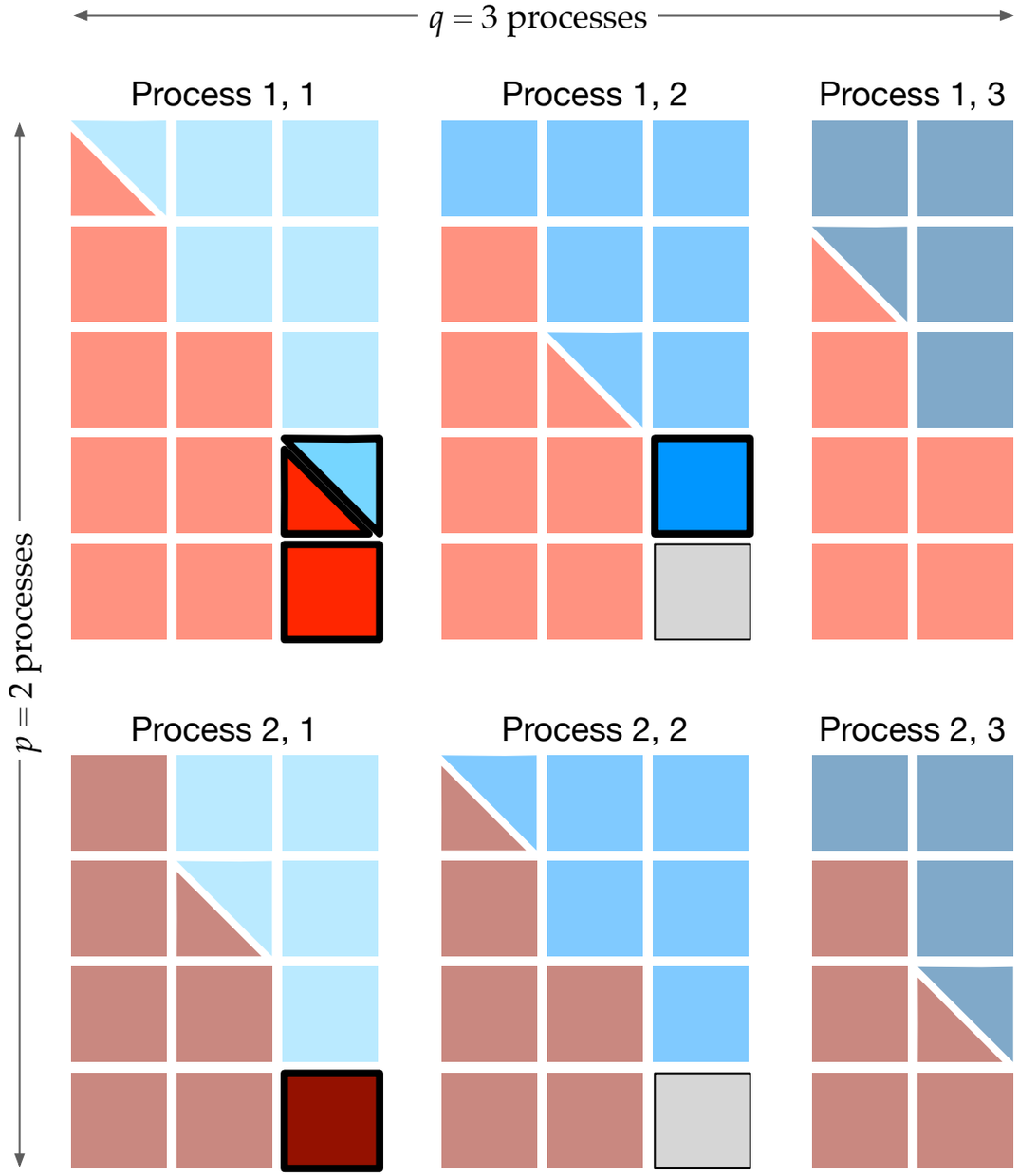
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

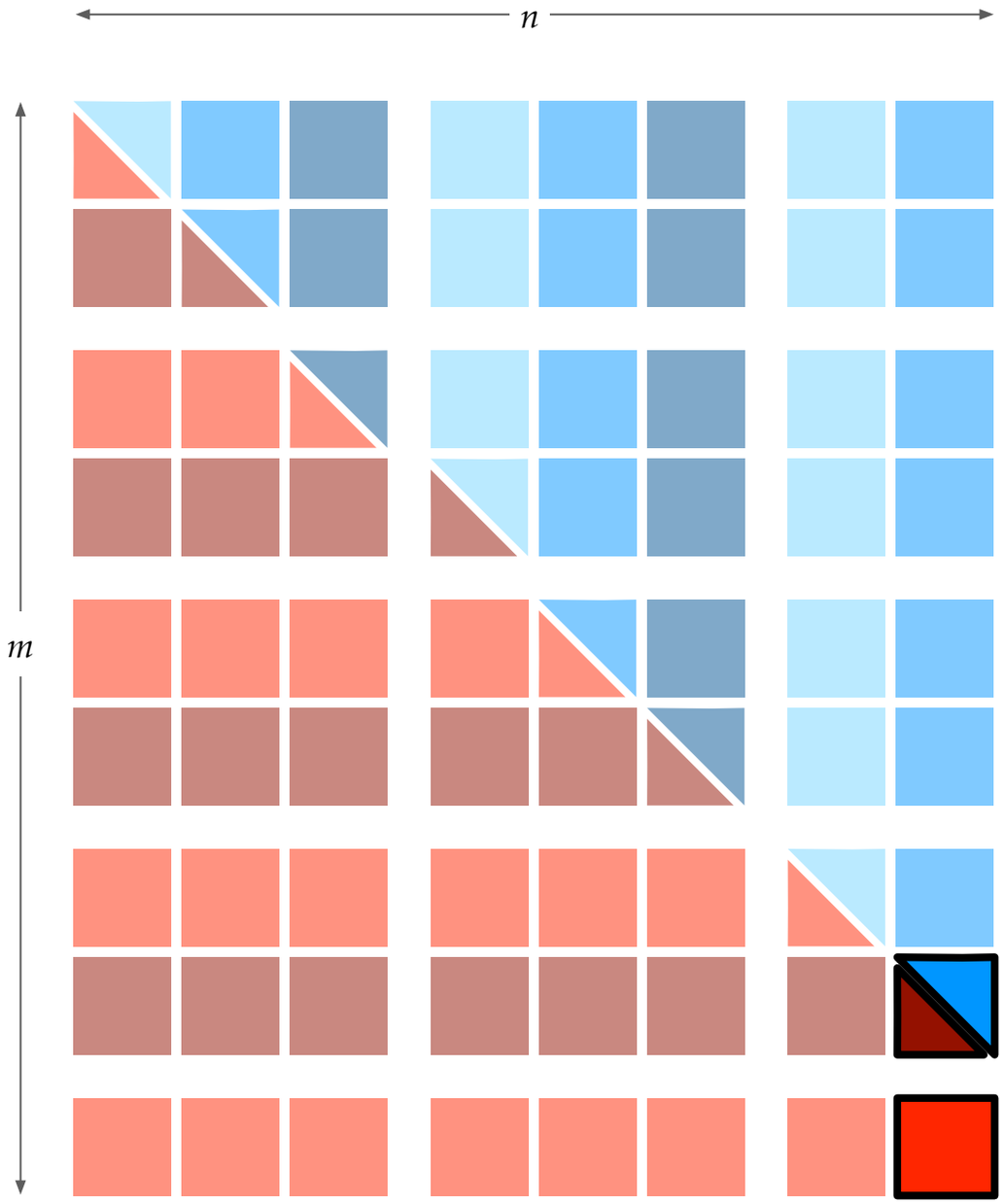


LU factorization

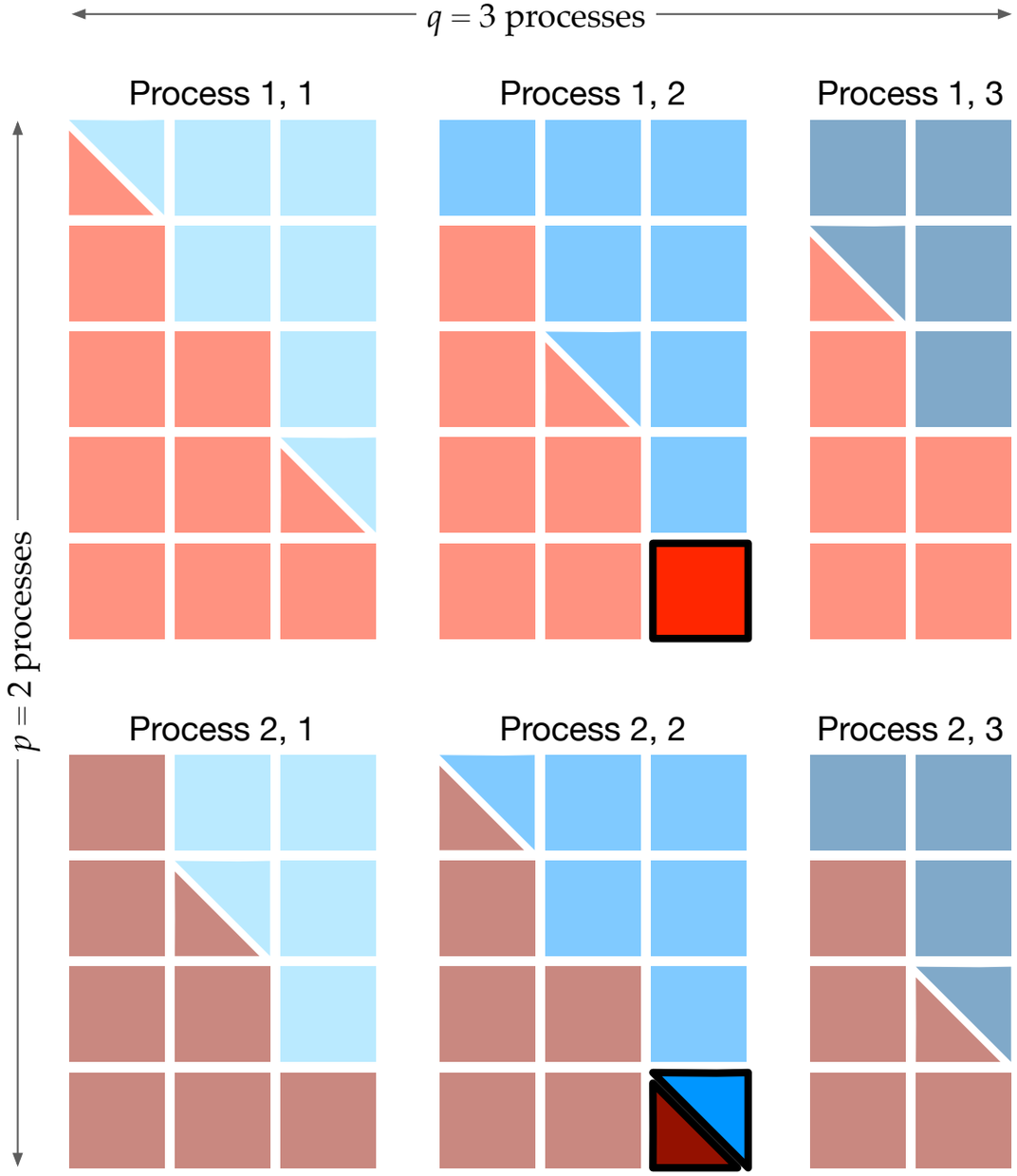
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view

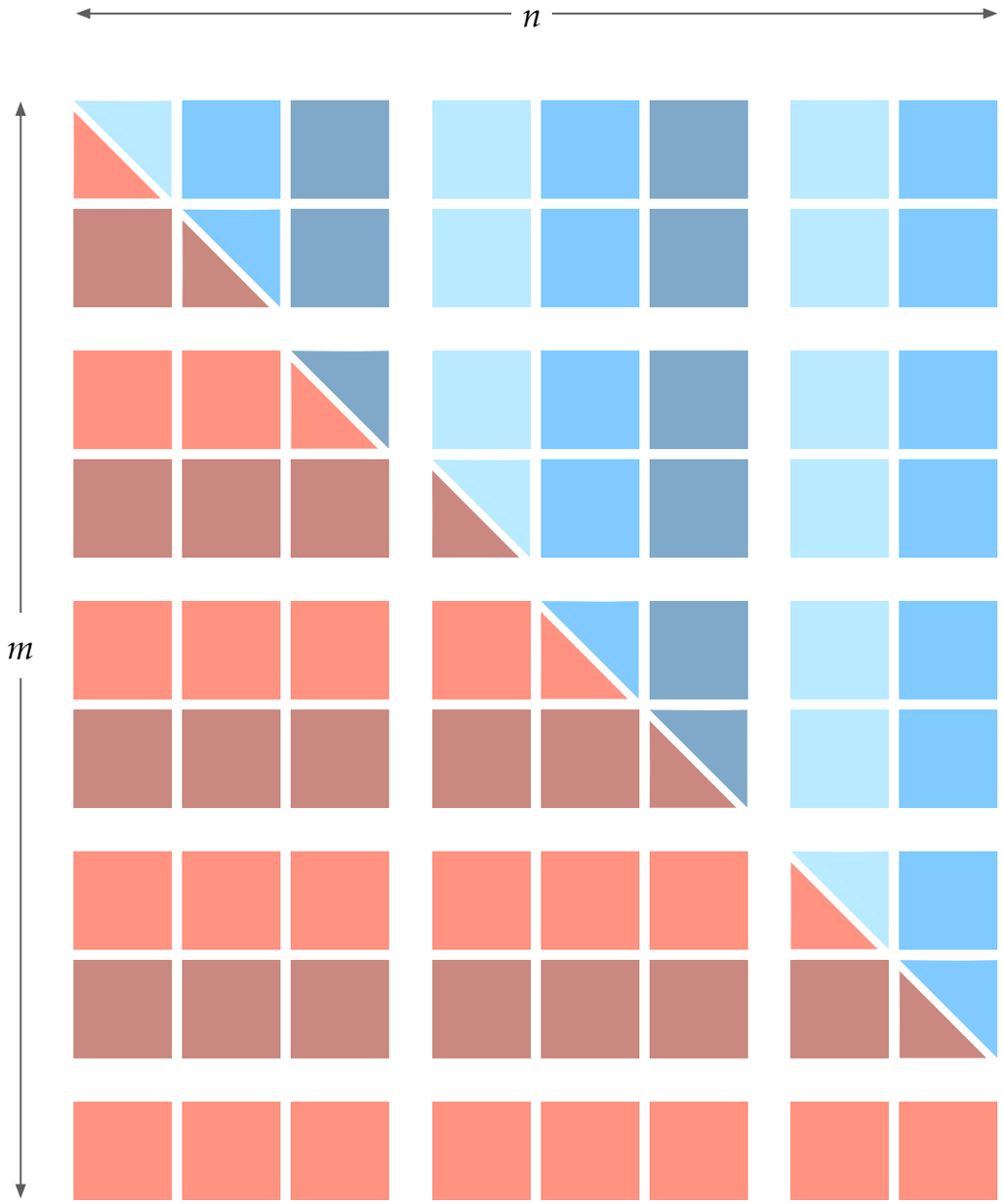


LU factorization

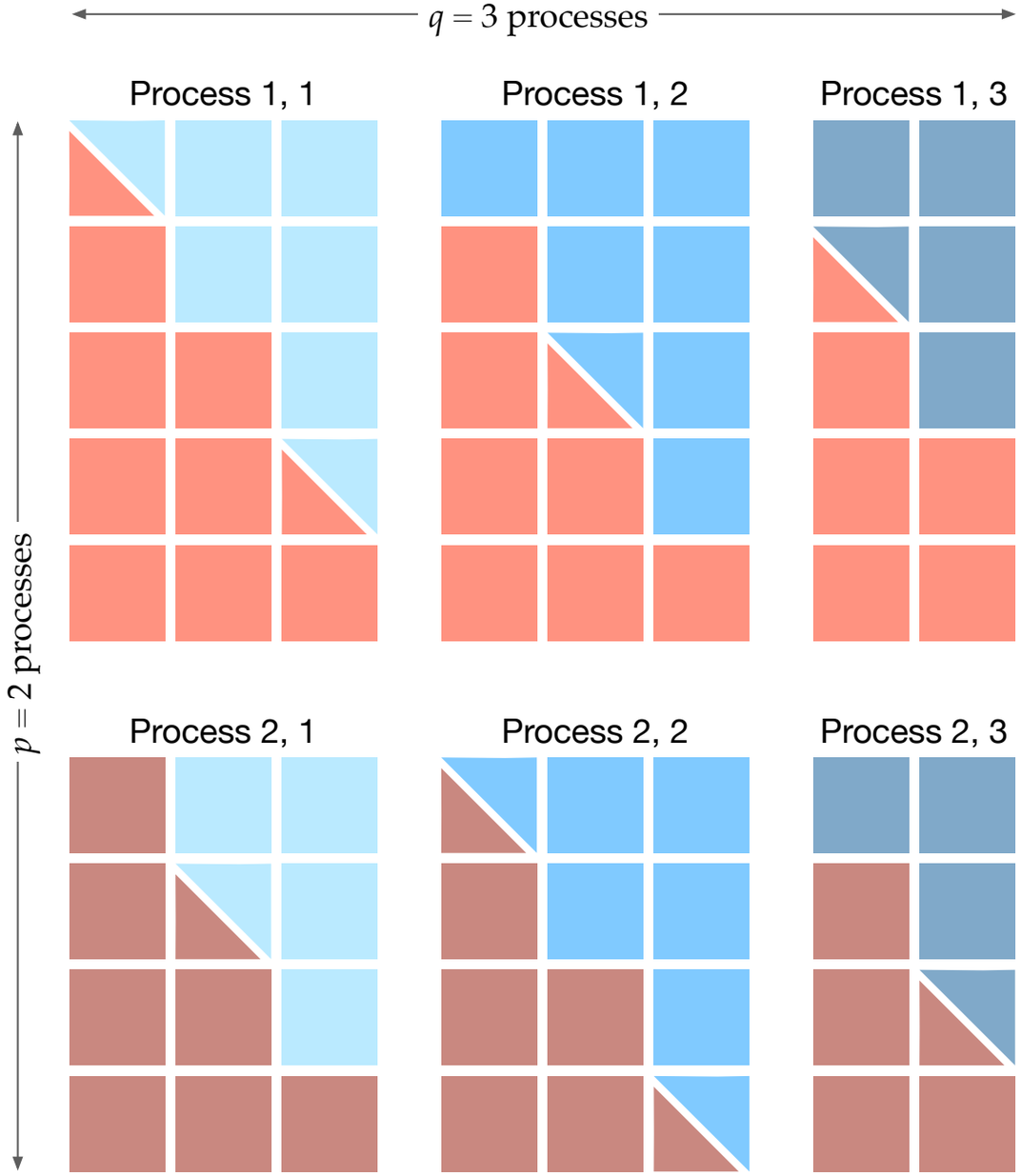
$m \times n$ matrix

$p \times q = 2$ x 3 process grid

Global matrix view



Local process point of view



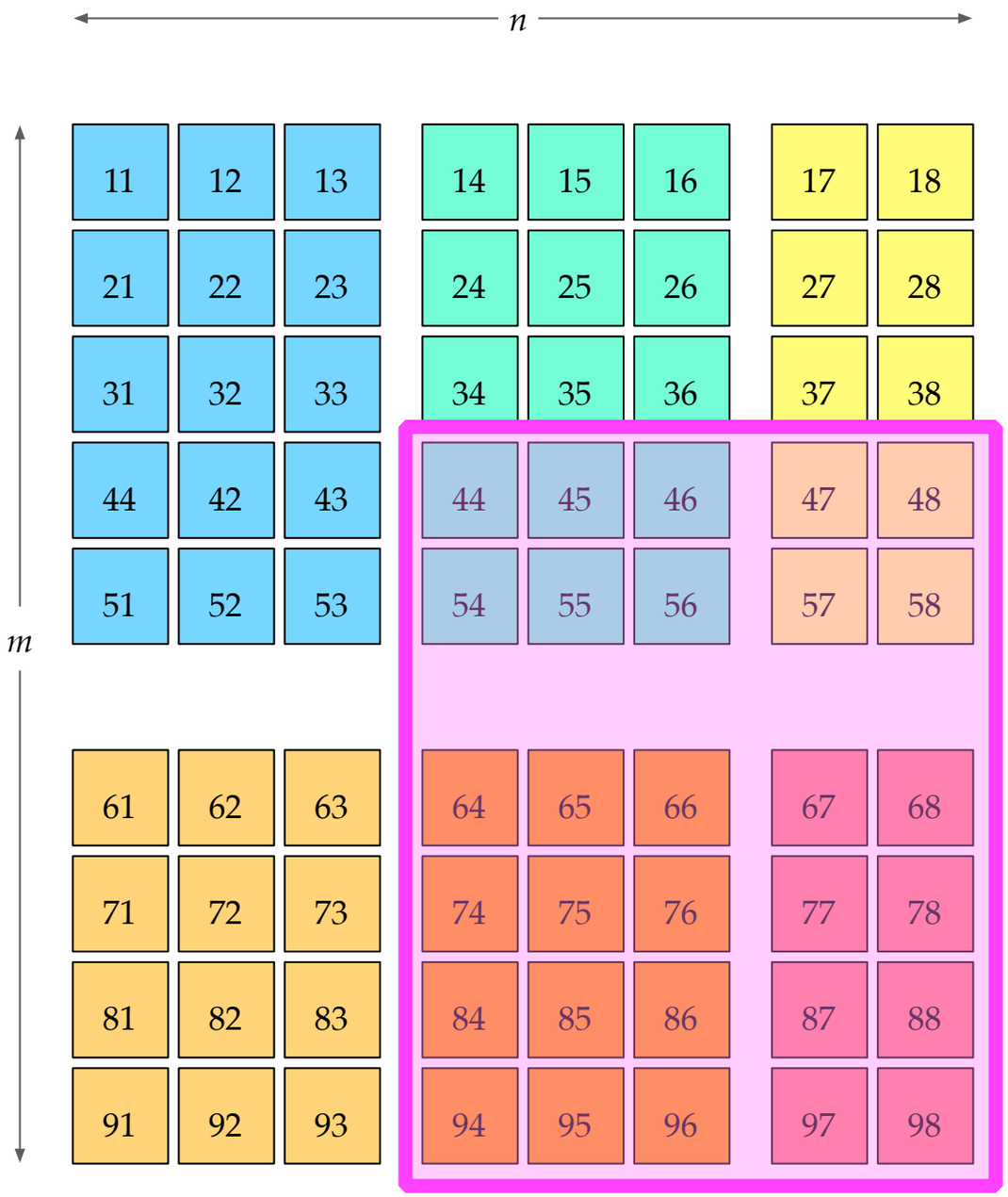
Why 2D block cyclic?

$m \times n$ matrix

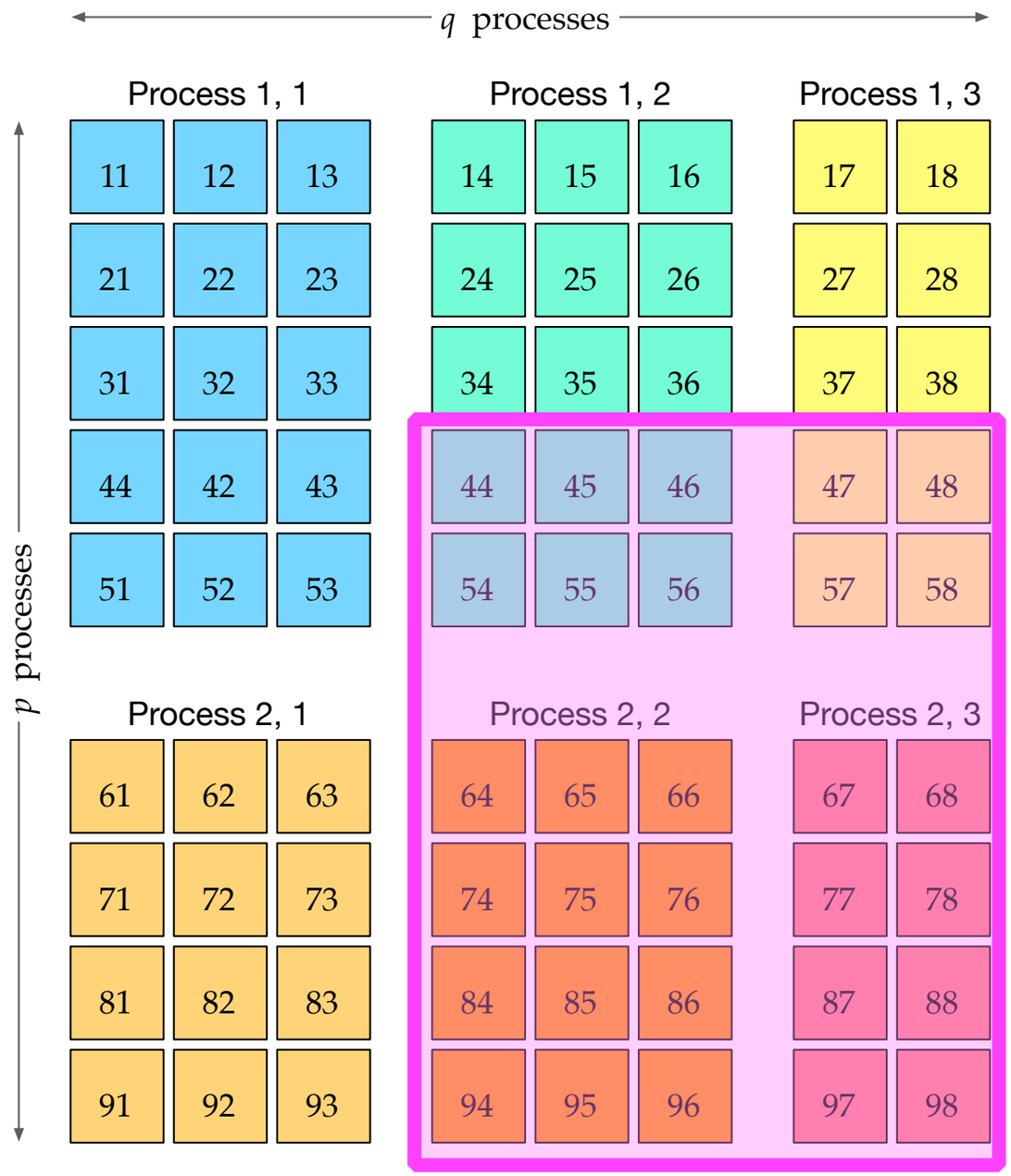
$p \times q = 2$ x 3 process grid

- Why not simple 2D distribution?

Global matrix view



Local process point of view



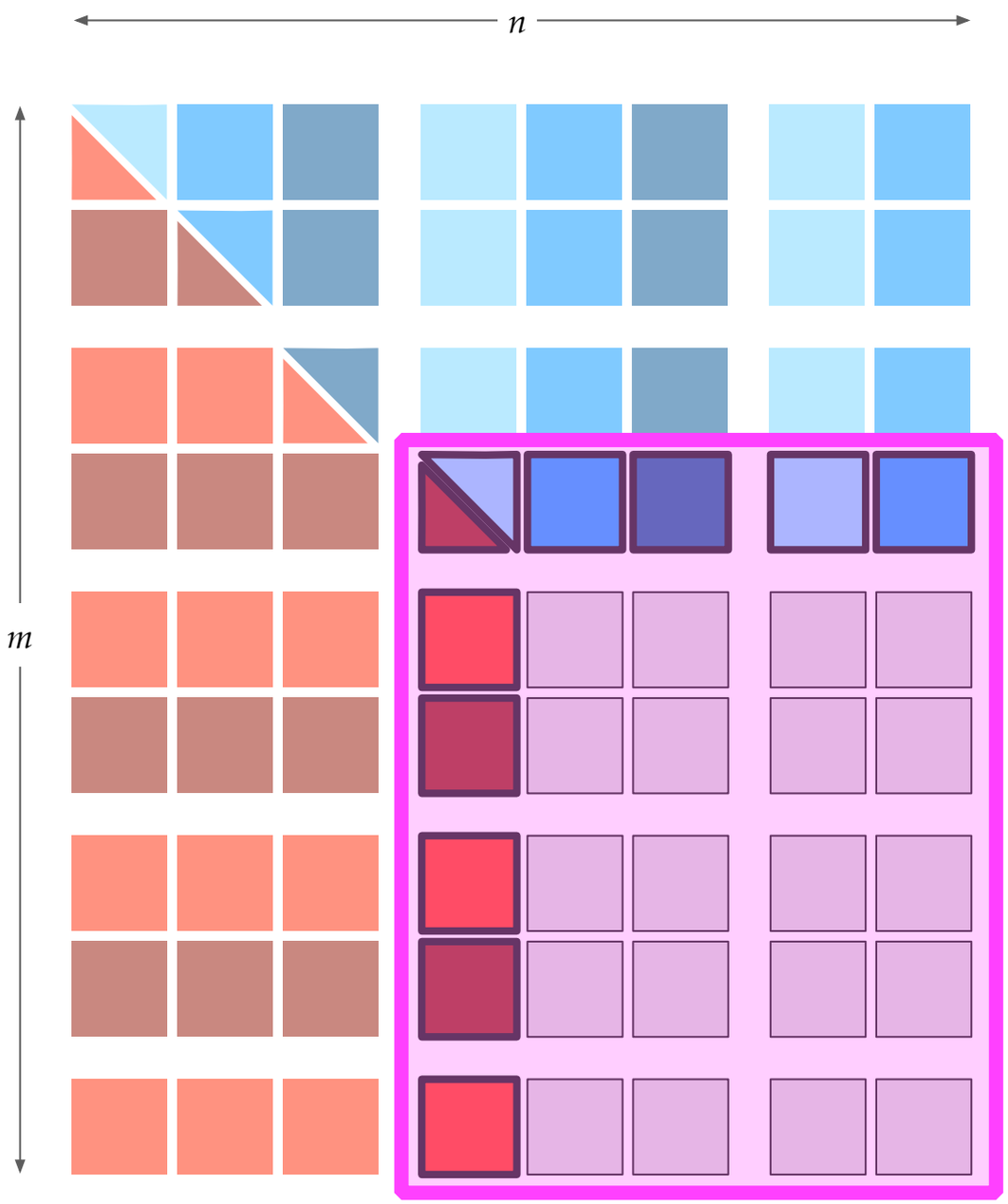
Why 2D block cyclic?

$m \times n$ matrix

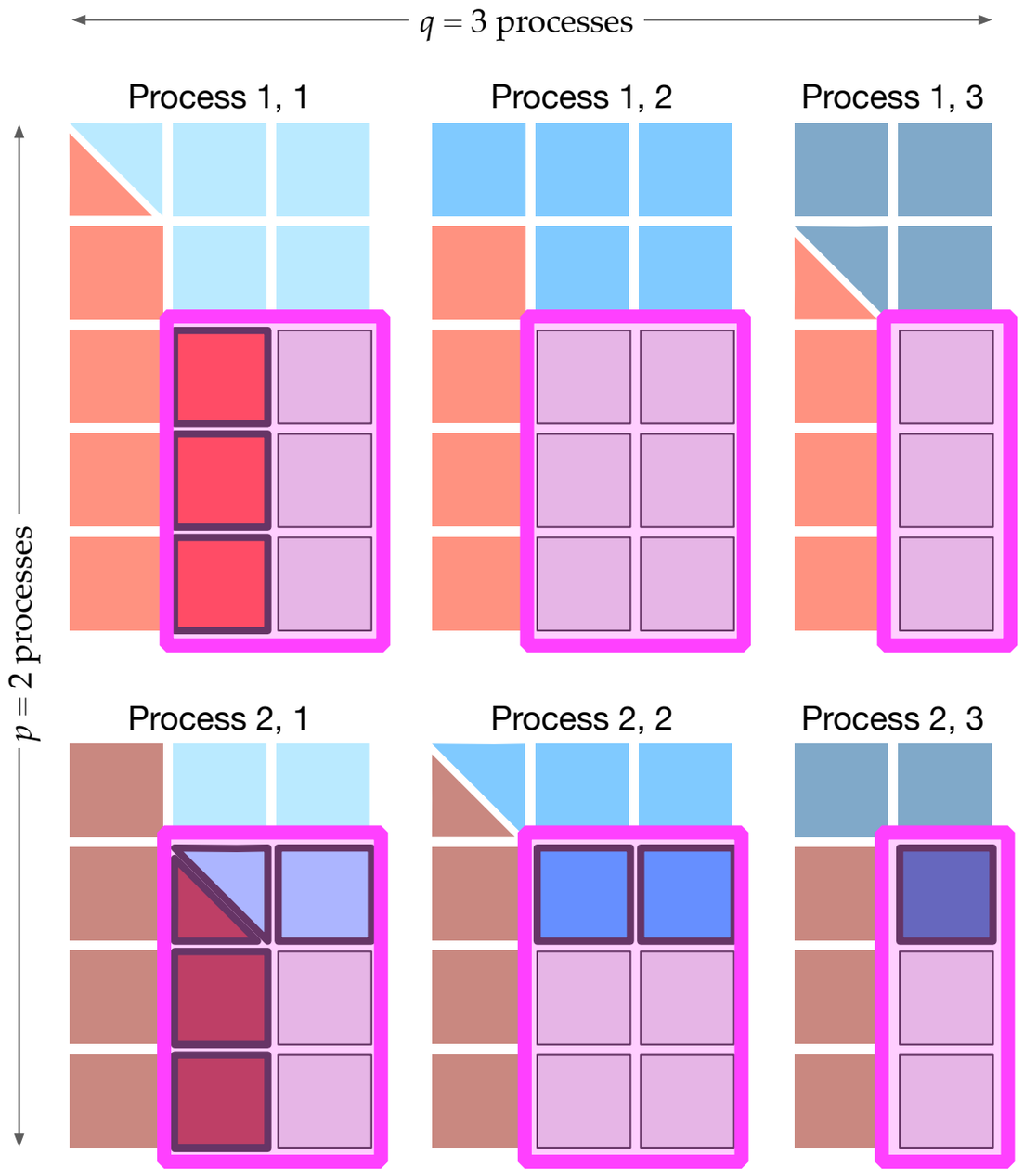
$p \times q = 2 \times 3$ process grid

- Better load balancing!

Global matrix view

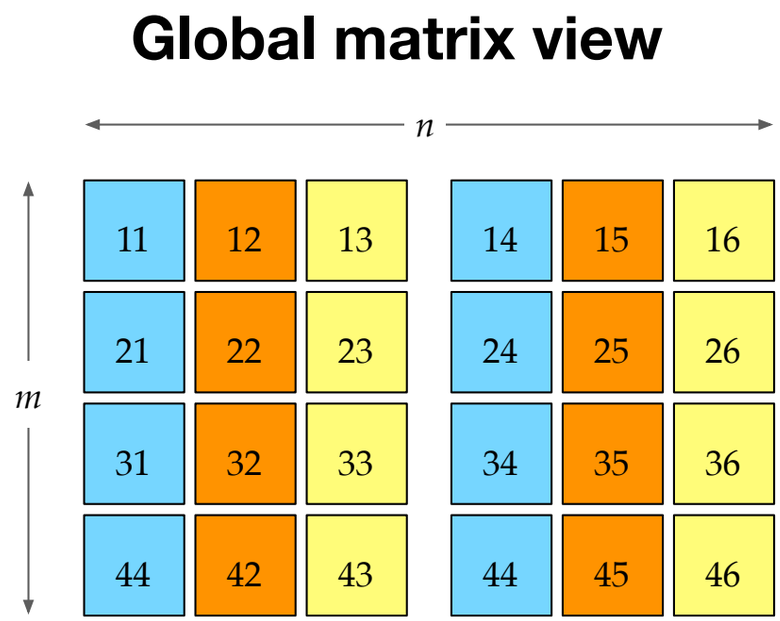


Local process point of view

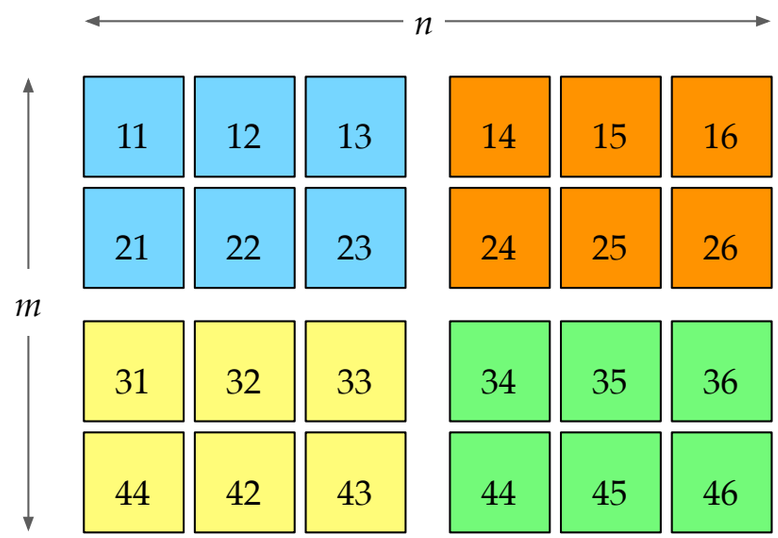


Other distributions

- 2D block cyclic not always required or best
- 1D block column cyclic (let $p = 1$) or
  1D block row cyclic (let $q = 1$)
  - MAGMA and SLATE use for distributing across multiple GPUs within node



- 1D and 2D block (not cyclic)
  - Fine if matrix doesn't shrink during computation
  - gemm
  - Jacobi SVD

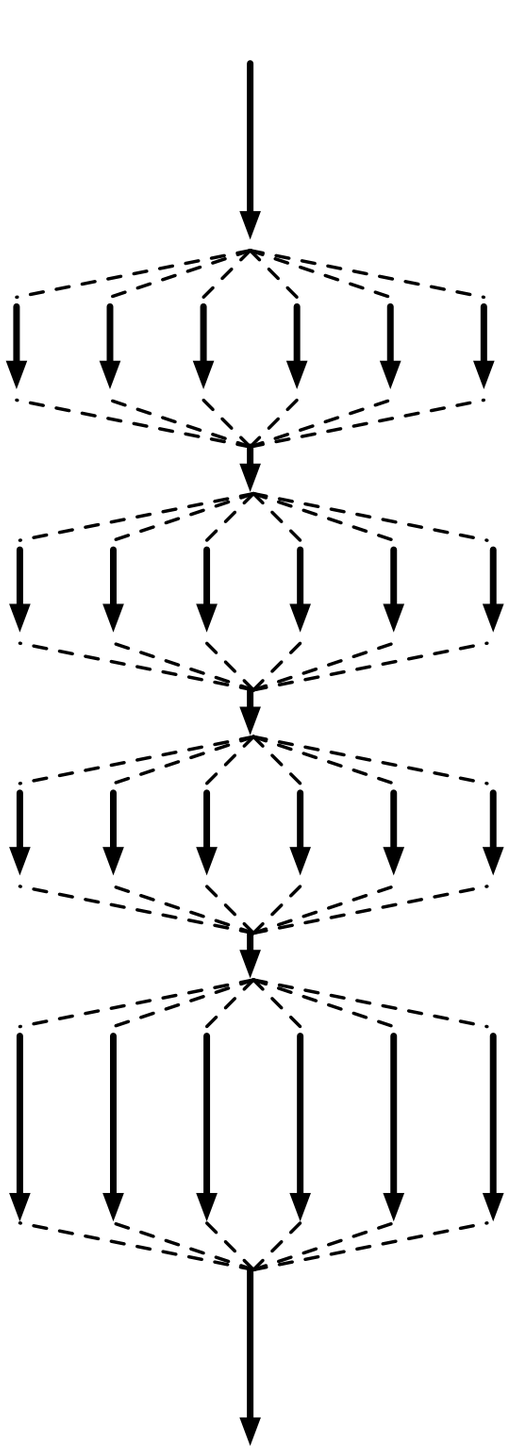


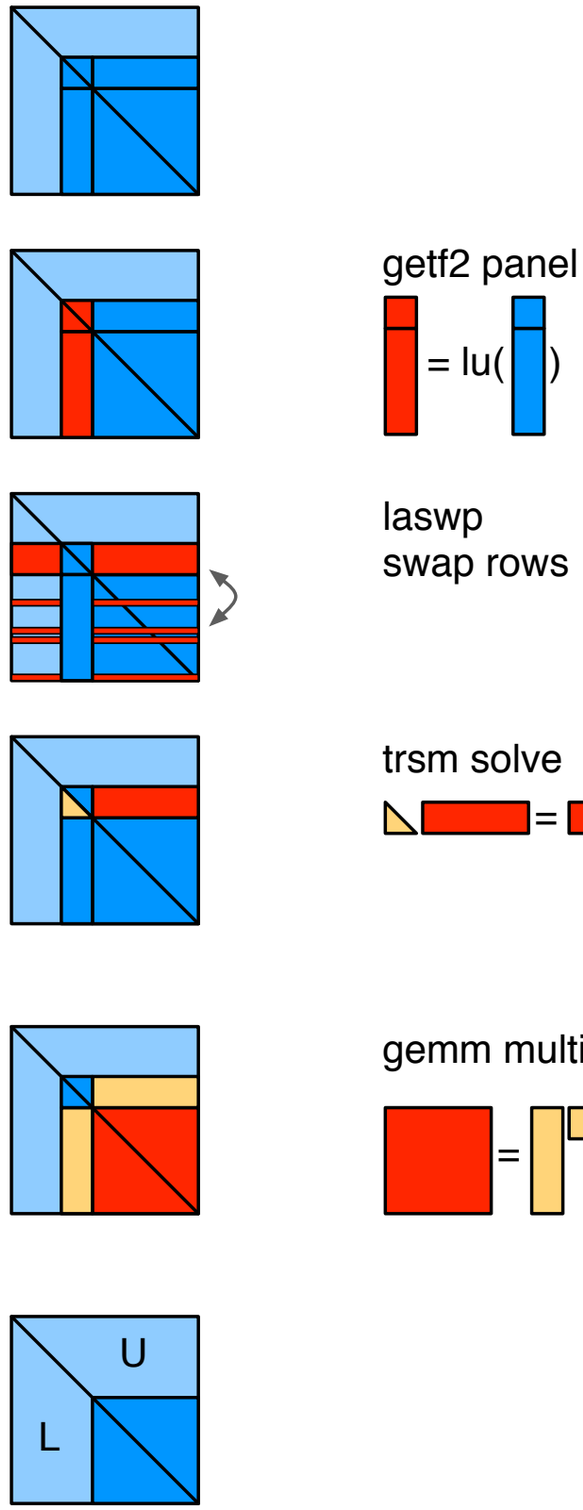
ScaLAPACK routines

- ~140–160 routines x 4 data types (s, d, c, z)
- **Driver routines**: solve an entire problem
  - `sv` Solve linear system: $Ax = b$
    - `gesv` general non-symmetric (LU)
    - `posv` symmetric positive definite (Cholesky)
    - ~~`sysv` symmetric indefinite (LDL^T)~~
    - Also ~~packed~~, banded, tridiagonal storage
  - `ls` Linear least squares: $Ax \cong b$
    - ~~`gglse` linear equality-constrained least squares~~
    - ~~`ggglm` general Gauss-Markov linear model~~
  - `ev` Eigenvalue decomposition: $Ax = \lambda x$ and $Ax = \lambda \mathrm{M} x$
    - `syevd` symmetric and symmetric generalized
    - ~~`geev`~~ non-symmetric (only gehrd Hessenberg reduction, no geev)
  - `svd` Singular value decomposition: $A = U\Sigma V^H$
    - `gesvd` standard ~~and generalized~~ (also not optimized for tall matrices)
    - ~~`gesdd` D&C (faster)~~

Parallelism in ScaLAPACK

- Similar to LAPACK
- Bulk-synchronous
- Most flops in gemm update
  - 2/3 n^3 term
  - Can use **sequential BLAS**, p x q = # cores = # MPI processes, num_threads = 1
  - Or **multi-threaded BLAS**, p x q = # nodes = # MPI processes, num_threads = # cores/node





getf2 panel

= lu( )

laswp
swap rows

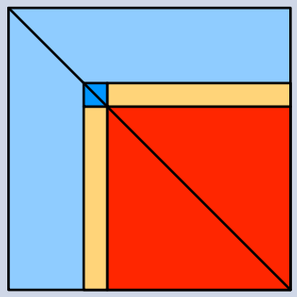
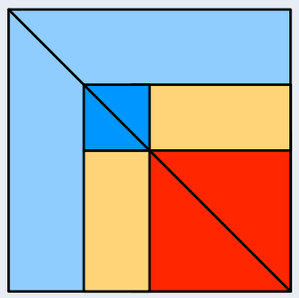
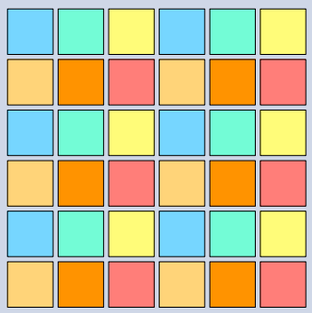
trsm solve

=

gemm multiply

= +

Legacy software libraries

	**LINPACK** (70s) vector operations	Level 1 BLAS
	**LAPACK** (80s) block operations	Level 3 BLAS
	**ScaLAPACK** (90s) 2D block cyclic distribution	PBLAS BLACS MPI

Emerging software solutions

- **PLASMA**
  - Tile layout & algorithms
  - Dynamic scheduling — OpenMP 4

- **MAGMA**
  - Hybrid multicore + accelerator (GPU, Xeon Phi)
  - Block algorithms (LAPACK style)

- **DPLASMA** — PaRSEC
  - Distributed with accelerators
  - Tile layout & algorithms
  - Dynamic scheduling — parameterized task graph

- **SLATE**
  - Distributed with accelerators
  - Tile layout & algorithms
  - Dynamic scheduling — OpenMP 4

2007 2009 2011 2017

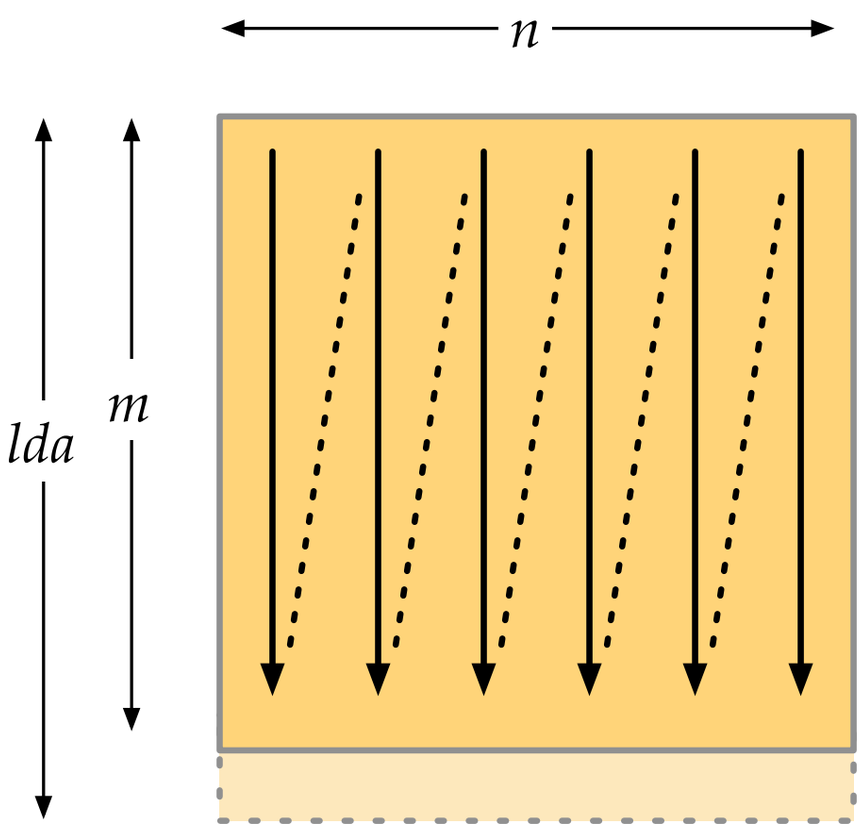
Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

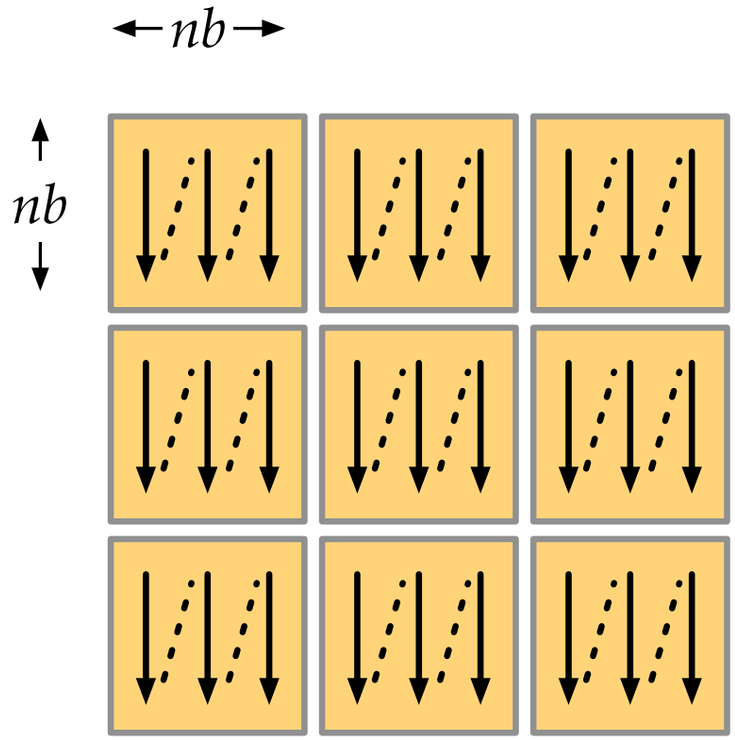
- New Software
  - **PLASMA using OpenMP**
  - DPLASMA and PaRSEC
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

Tile matrix layout

LAPACK column major



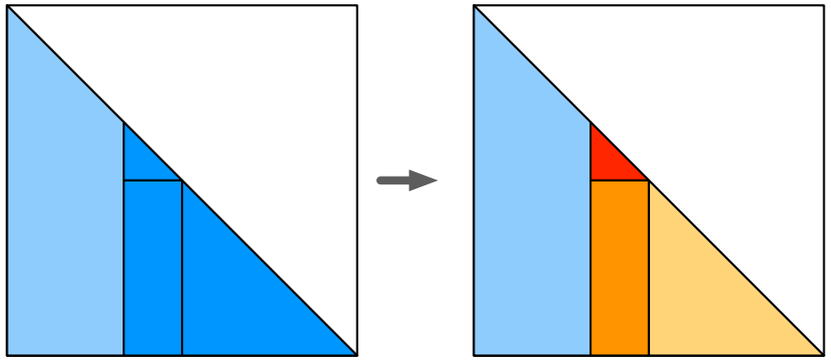
(D)PLASMA tile layout

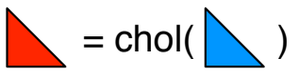


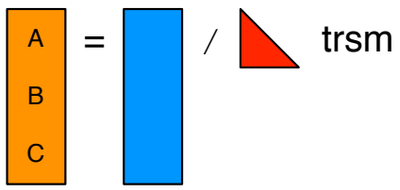
- Tiled layout
  - Each tile is contiguous (column major)
  - Enables dataflow scheduling
  - Cache and TLB efficient (reduces conflict misses and false sharing)
  - MPI messaging efficiency (zero-copy communication)
  - In-place, parallel layout translation

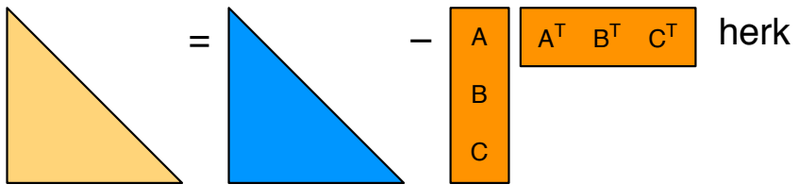
Tile algorithms: Cholesky

LAPACK Algorithm (right looking)

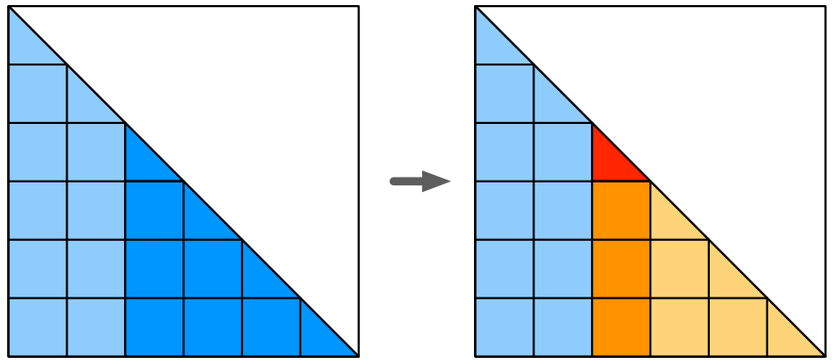


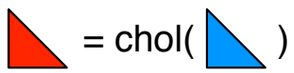


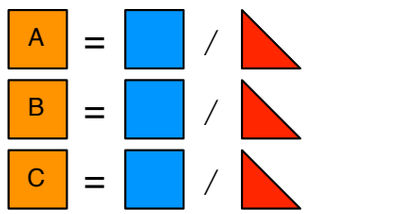


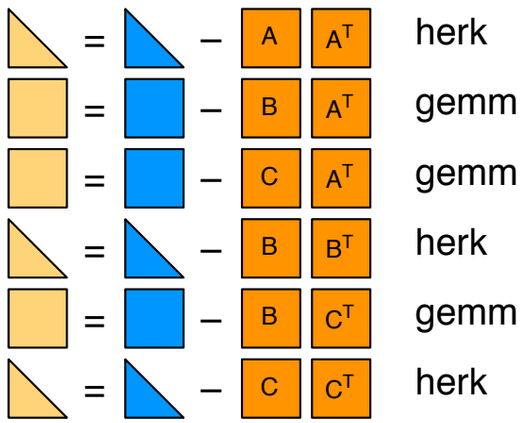


Tile Algorithm

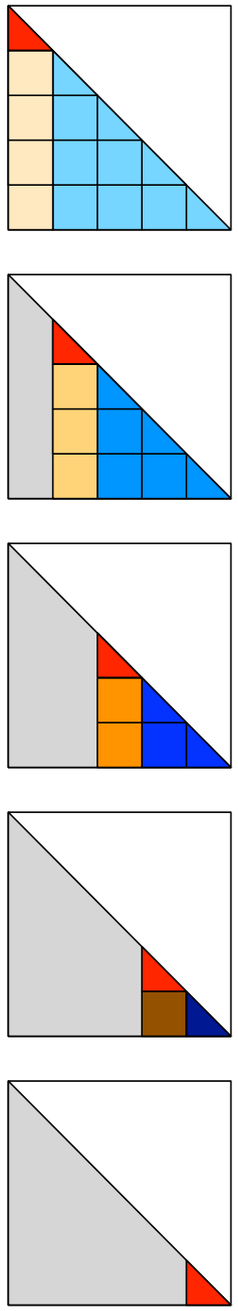






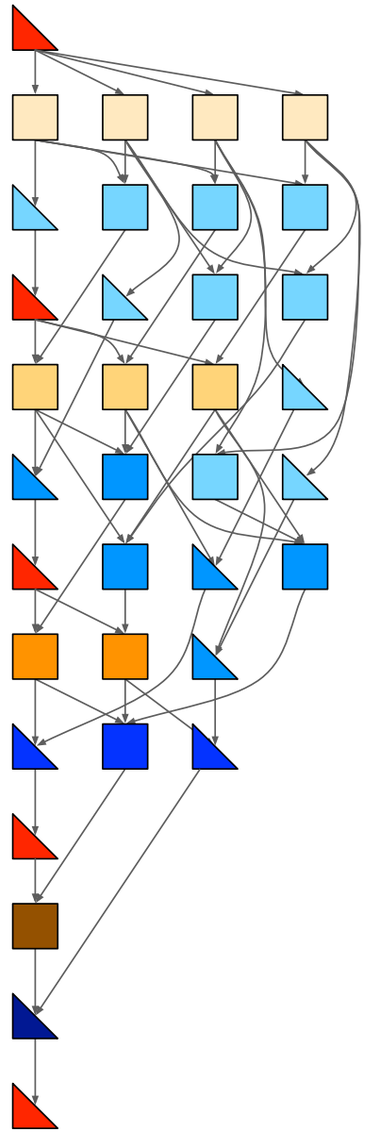


Track dependencies — Directed acyclic graph (DAG)



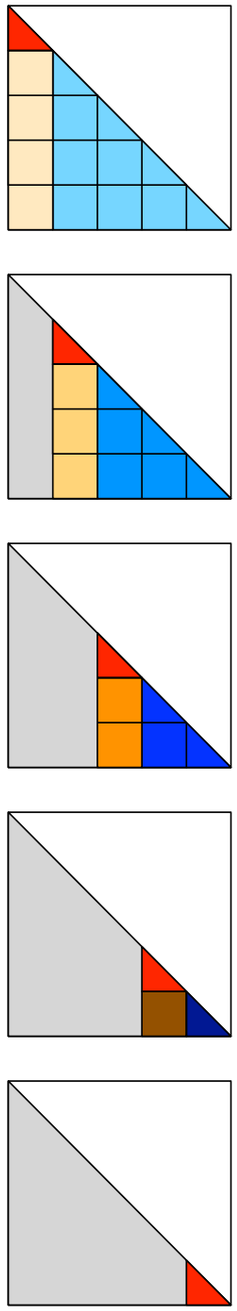
Fork-join schedule on 4 cores
with artificial synchronizations

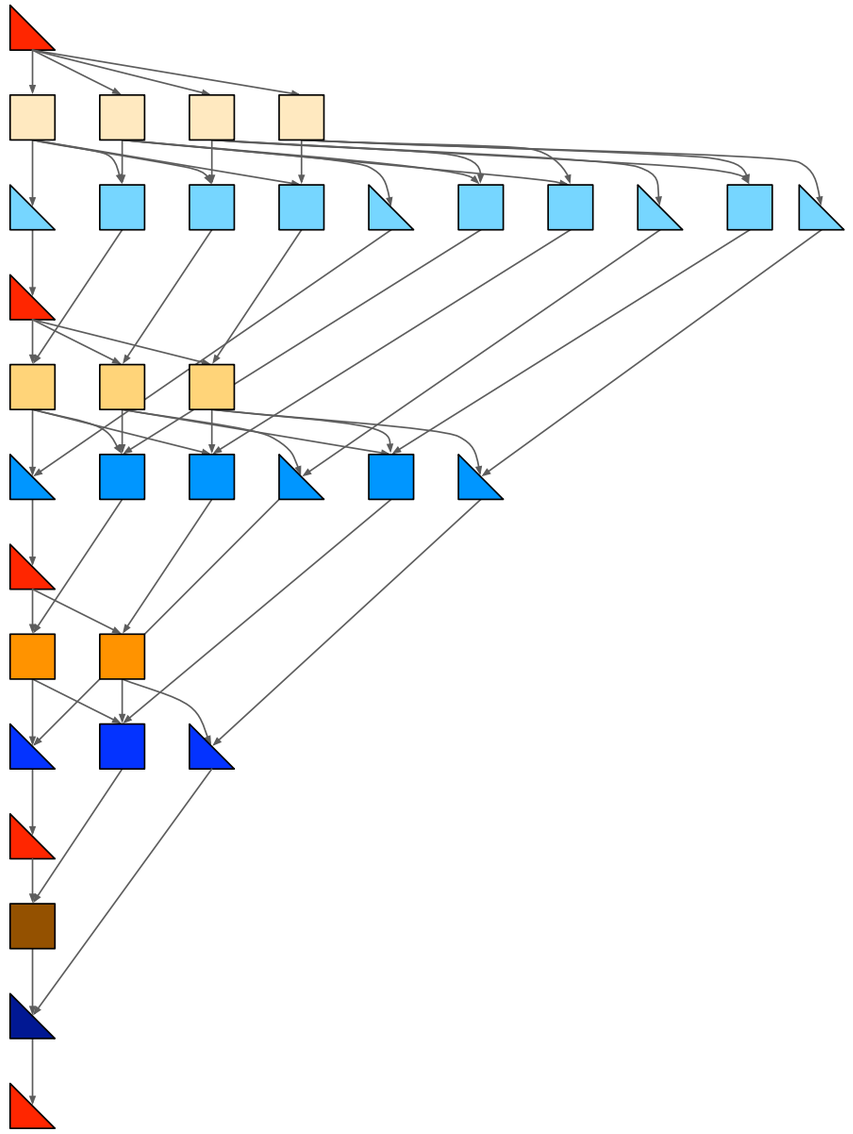
Reordered without
synchronizations



Track dependencies — Directed acyclic graph (DAG)

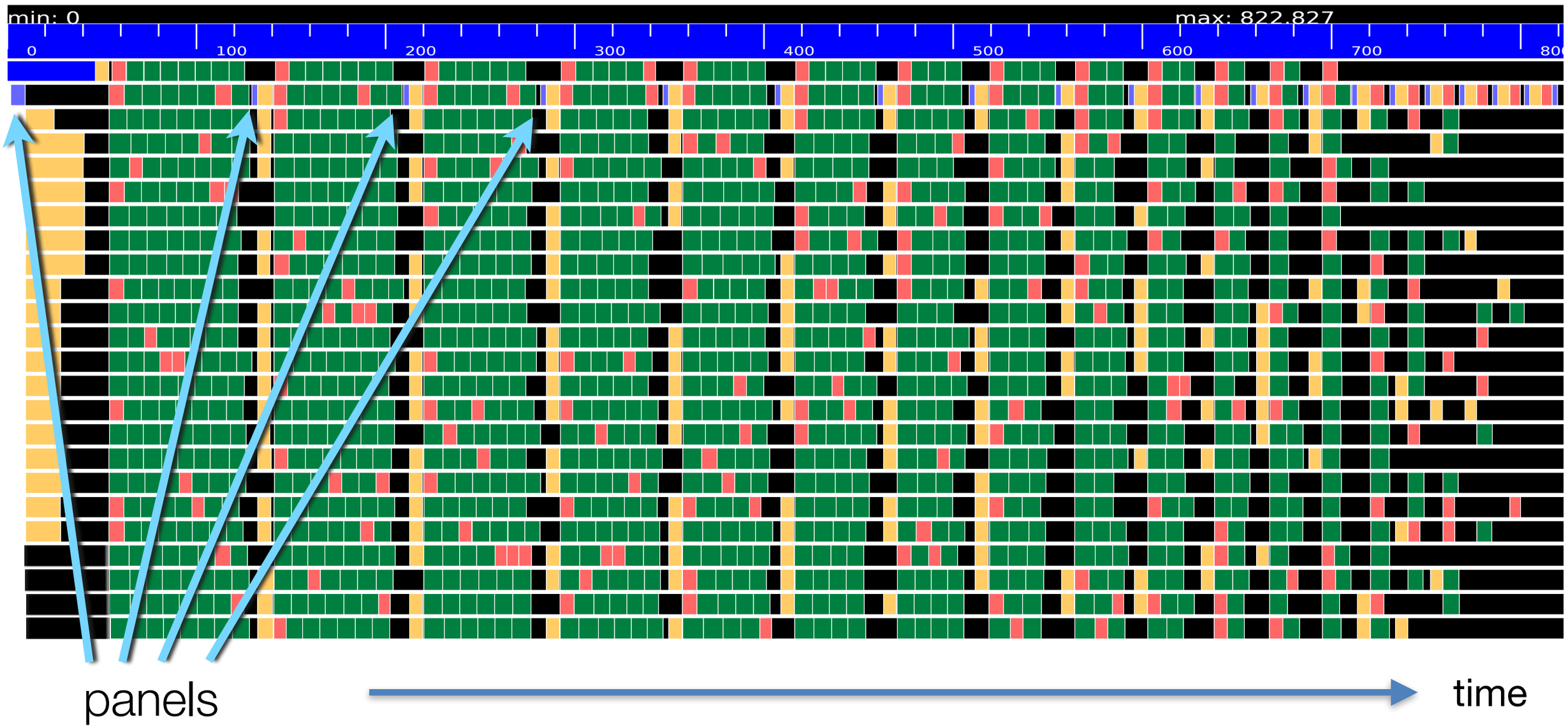
Critical path with infinite number of processors (need 10 here) yields minimum possible time





Execution trace

- LAPACK-style fork-join leave cores idle



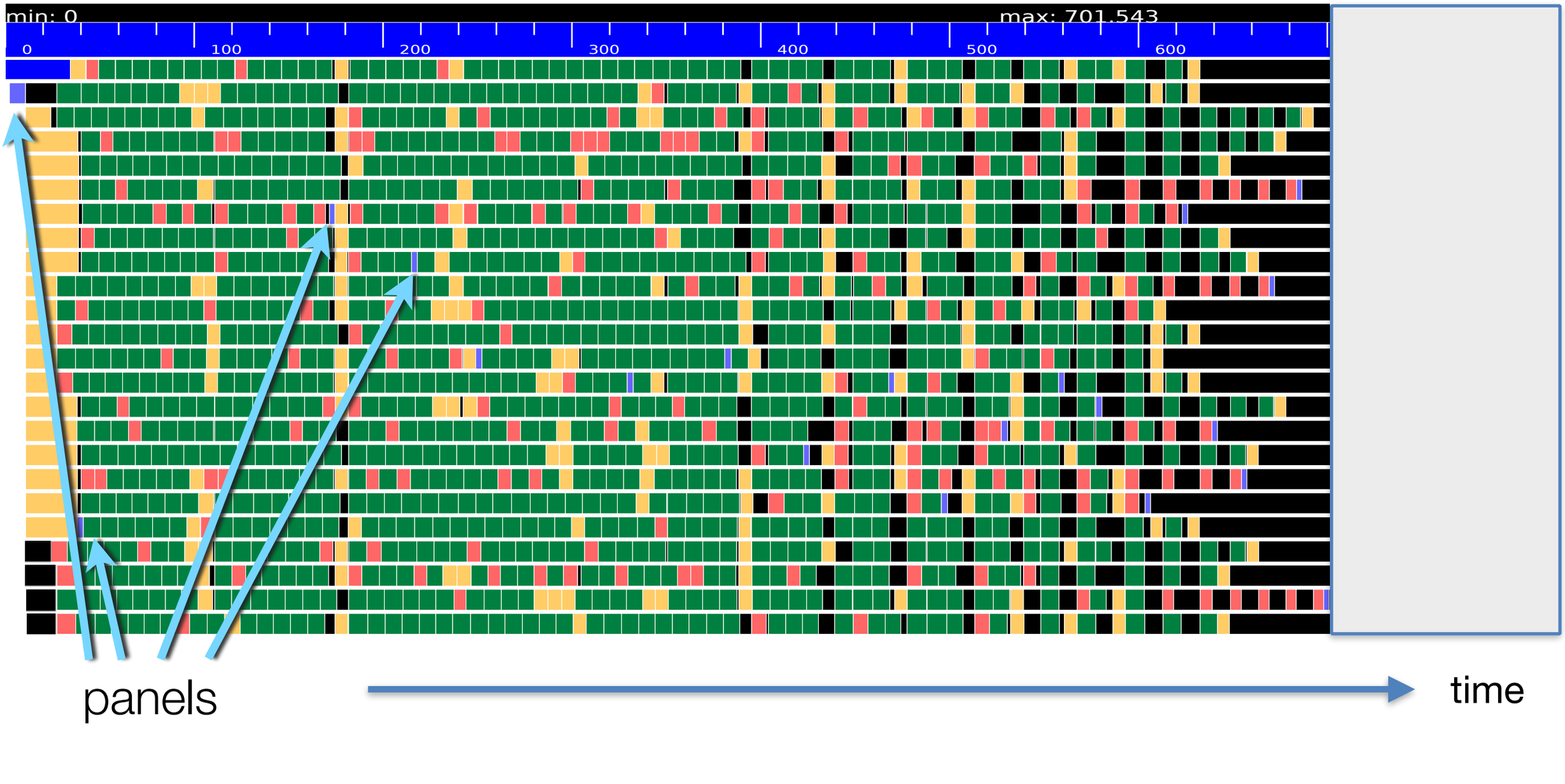
24 cores
Matrix is 8000 x 8000, tile size is 400 x 400.

potrf | trsm | syrk | gemm | idle

ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

Execution trace

- PLASMA squeezes out idle time



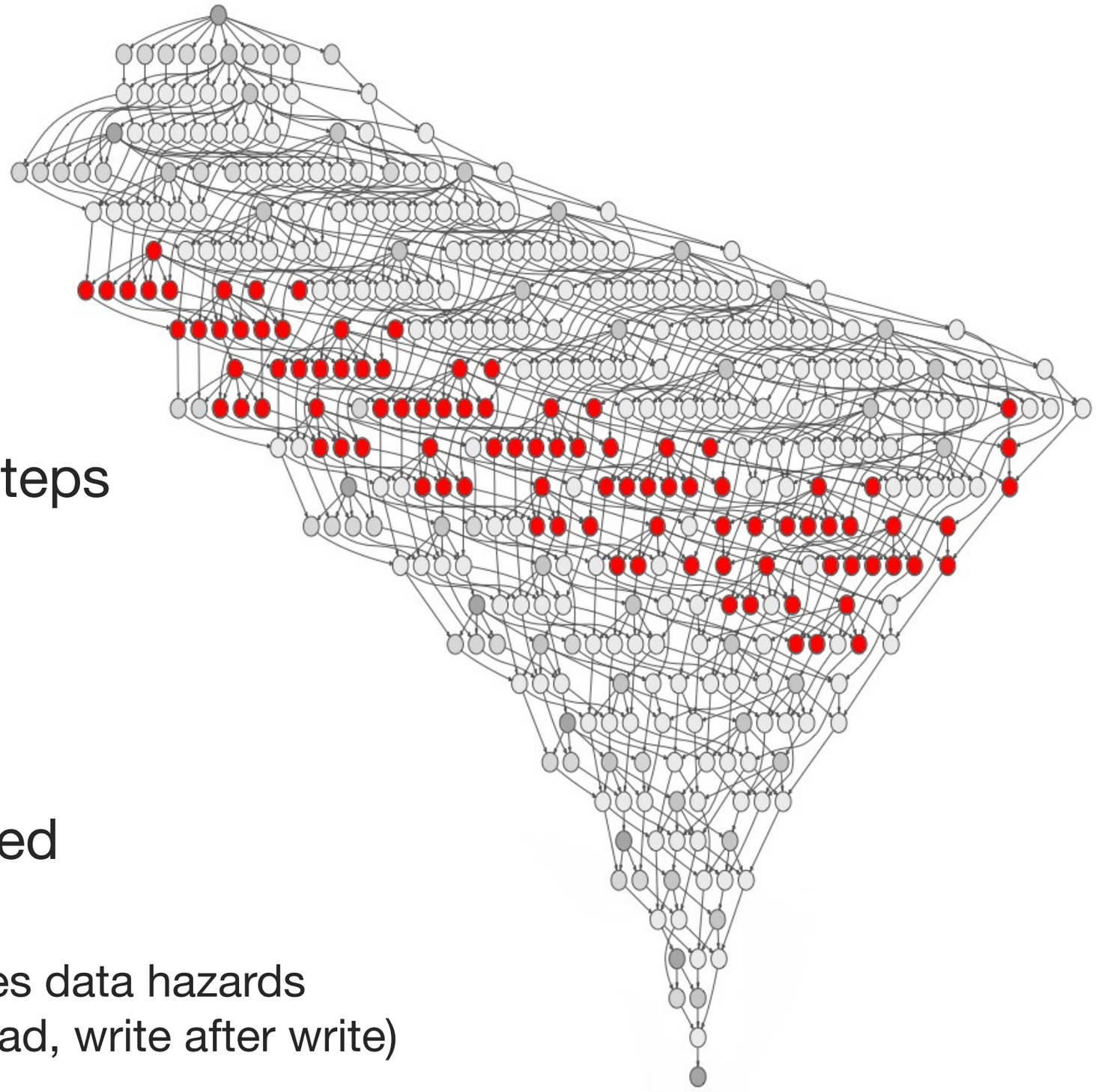
24 cores
Matrix is 8000 x 8000, tile size is 400 x 400.

potrf trsm syrk gemm idle

ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

Dataflow scheduling

- Exploit parallelism
- Load balance
- Remove artificial synchronization between steps
- Maximize data locality
- Parallel correctness inherited from serial tile algorithm
  - Runtime automatically resolves data hazards (read after write, write after read, write after write)

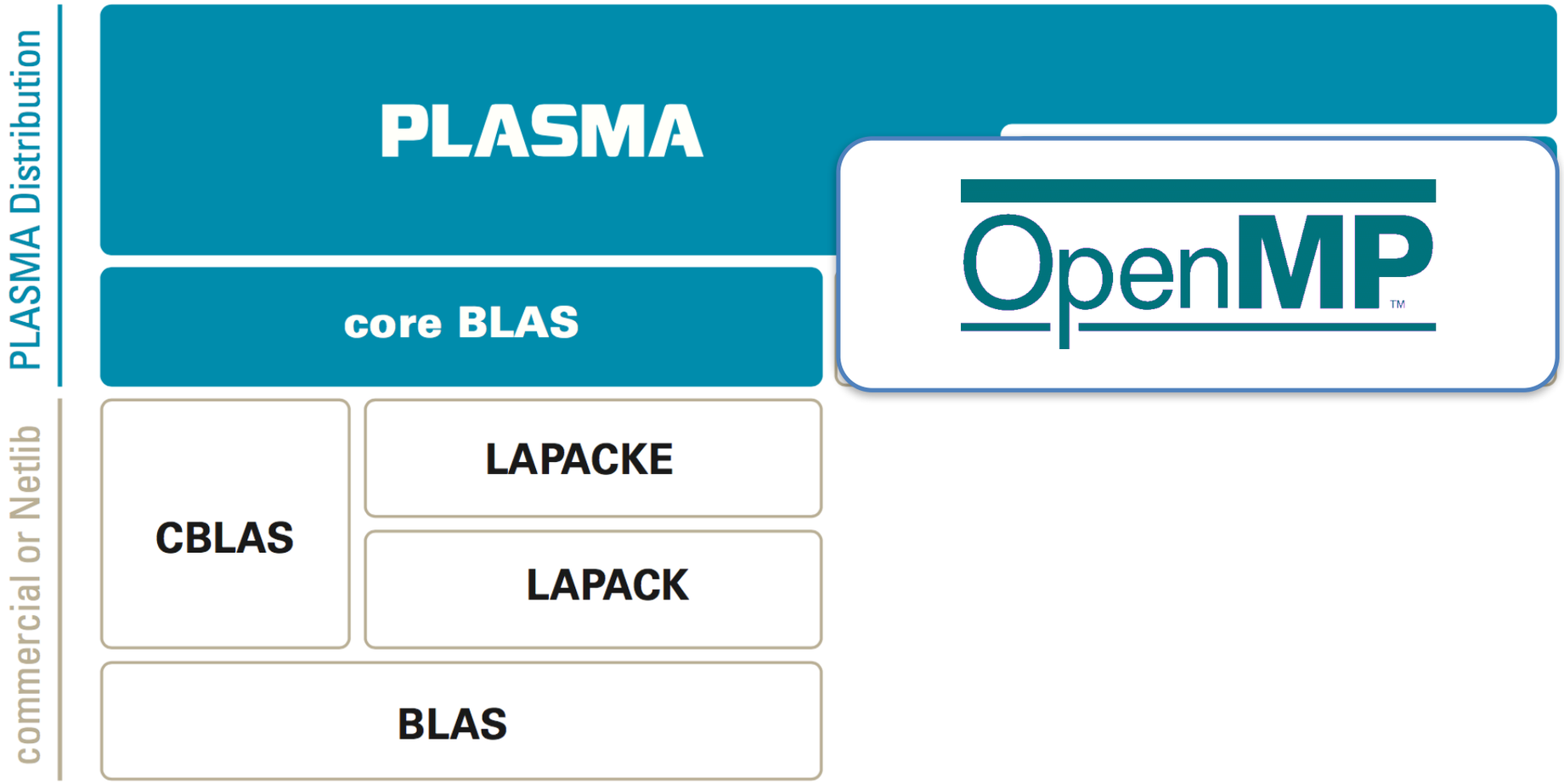


Dynamic scheduling runtimes

- Jade — Stanford University
- SMPSs / OMPSs — Barcelona Supercomputer Center
- StarPU — INRIA Bordeaux
- QUARK — University of Tennessee
- SuperGlue & DuctTEiP — Uppsala University
- OpenMP 4

May 2008	OpenMP 3.0	→	#pragma omp task
April 2009		GCC 4.4	
July 2013	OpenMP 4.0	→	#pragma omp task **depend**
April 2014		GCC 4.9	

PLASMA architecture



Cholesky: pseudo-code

```
// Sequential Tile Cholesky
for k = 1 ... ntiles {
    potrf( Akk )
    for i = k+1 ... ntiles {
        trsm( Akk, Aik )
    }
    for i = k+1 ... ntiles {
        syrk( Aik, Aii )
        for j = i+1 ... ntiles {
            gemm( Ajk, Aik, Aij )
        }
    }
}
```

```
// PLASMA OpenMP Tile Cholesky
for k = 1 .. ntiles {
    omp task potrf( ... )
    for i = k+1 .. ntiles {
        omp task trsm( ... )
    }
    for i = k+1 .. ntiles {
        omp task syrk( ... ) {
        for j = i+1 .. ntiles
            omp task gemm( ... )
        }
    }
}
```

Cholesky: OpenMP

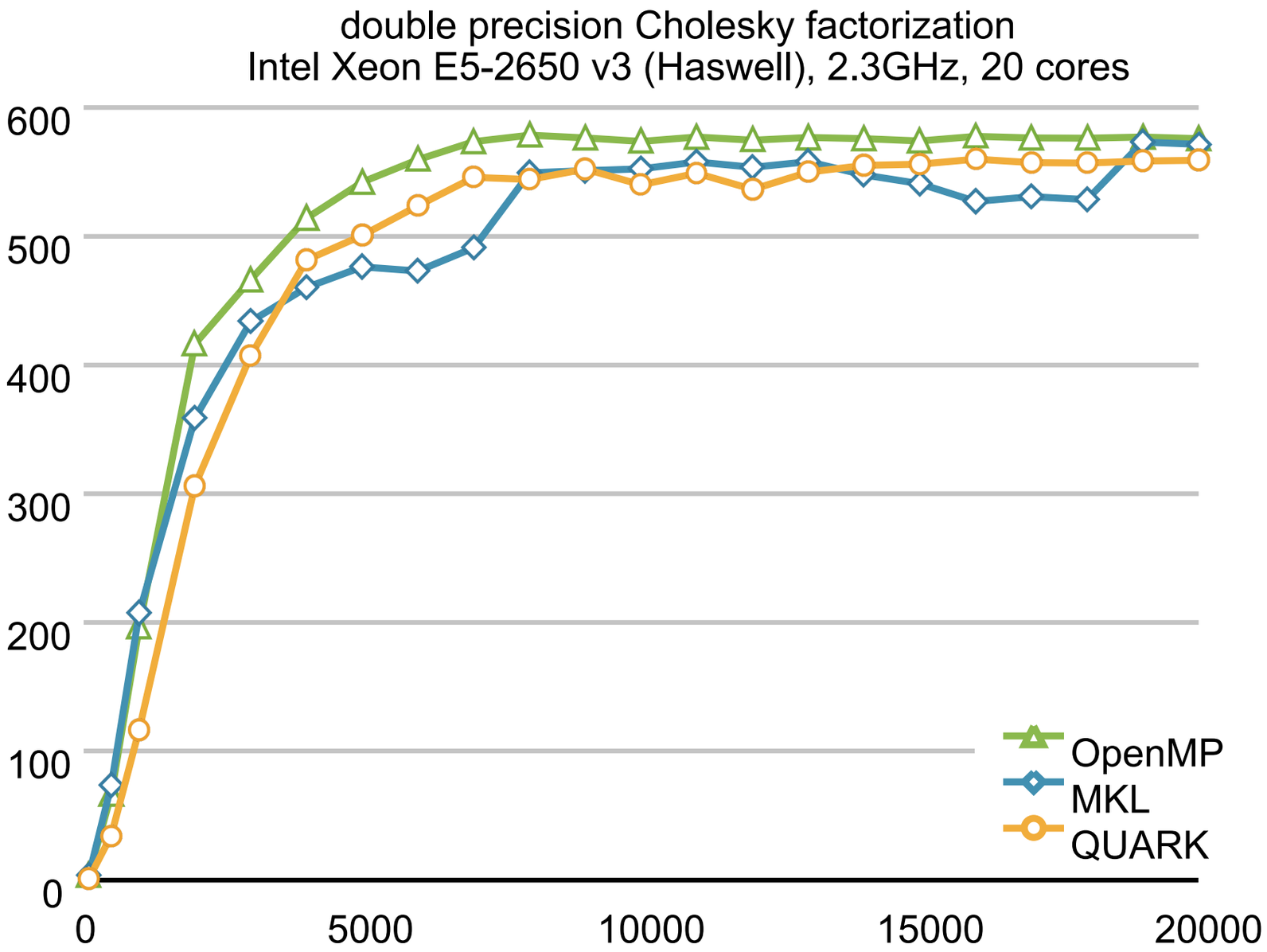
```
#pragma omp parallel
#pragma omp master
{
    for (k = 0; k < nt; k++) {
        #pragma omp task depend(inout:A(k,k)[0:nb*nb])
        info = LAPACKE_dpotrf_work(
                LAPACK_COL_MAJOR,
                lapack_const(PlasmaLower),
                nb, A(k,k), nb);

        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(k,k)[0:nb*nb]) \
                             depend(inout:A(m,k)[0:nb*nb])
            cblas_dtrsm(
                    CblasColMajor,
                    CblasRight, CblasLower,
                    CblasTrans, CblasNonUnit,
                    nb, nb,
                    1.0, A(k,k), nb,
                         A(m,k), nb);
        }
        for (m = k+1; m < nt; m++) {
            #pragma omp task depend(in:A(m,k)[0:nb*nb]) \
                             depend(inout:A(m,m)[0:nb*nb])
            cblas_dsyrk(
                    CblasColMajor,
                    CblasLower, CblasNoTrans,
                    nb, nb,
                    -1.0, A(m,k), nb,
                      1.0, A(m,m), nb);

            for (n = k+1; n < m; n++) {
                #pragma omp task depend(in:A(m,k)[0:nb*nb]) \
                                 depend(in:A(n,k)[0:nb*nb]) \
                                 depend(inout:A(m,n)[0:nb*nb])
                cblas_dgemm(
                        CblasColMajor,
                        CblasNoTrans, CblasTrans,
                        nb, nb, nb,
                        -1.0, A(m,k), nb,
                              A(n,k), nb,
                         1.0, A(m,n), nb);
            }
        }
    }
}
```

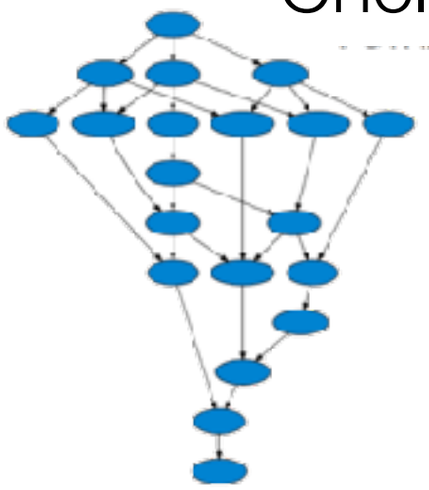
```
#pragma omp task depend(in:A(m,k)[0:nb*nb]) \
                 depend(in:A(n,k)[0:nb*nb]) \
                 depend(inout:A(m,n)[0:nb*nb])
cblas_dgemm(
        CblasColMajor,
        CblasNoTrans, CblasTrans,
        nb, nb, nb,
        -1.0, A(m,k), nb,
              A(n,k), nb,
         1.0, A(m,n), nb);
```

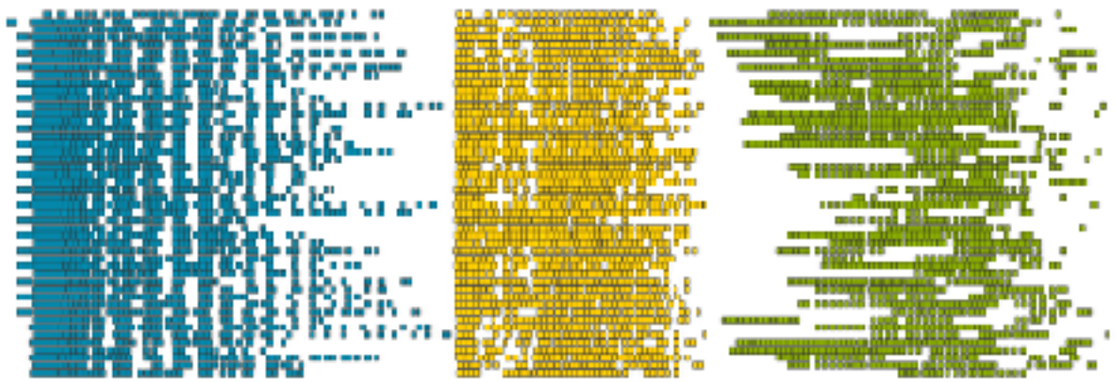
Cholesky performance



Merging DAGs

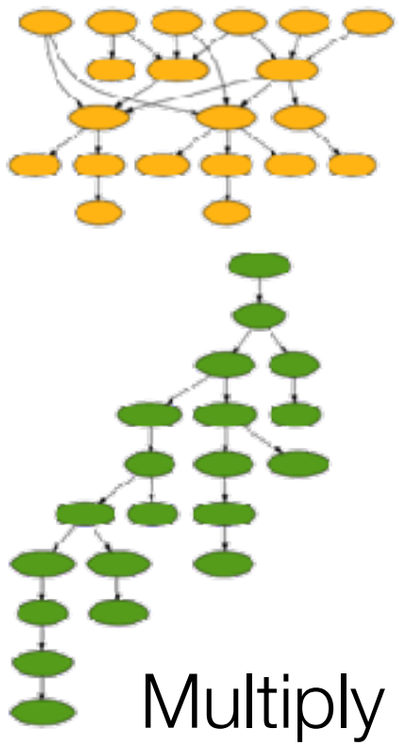
Cholesky $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$

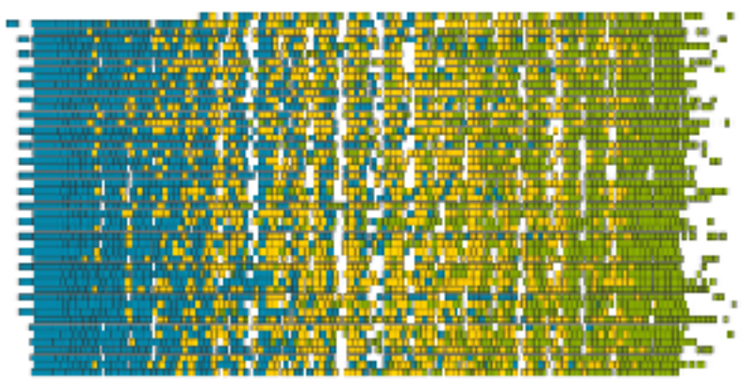




time →

Invert $\boldsymbol{L}^{-1}$



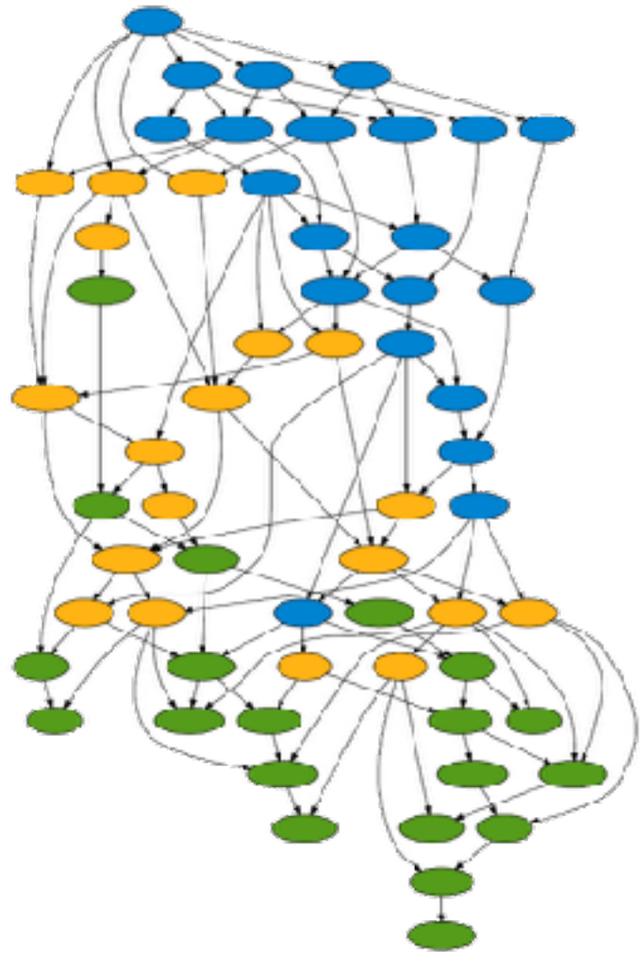


time →

Multiply $\boldsymbol{A}^{-1} = \boldsymbol{L}^{-T}\boldsymbol{L}^{-1}$

48 cores, matrix is 4000 x 4000, tile size is 200 x 200.

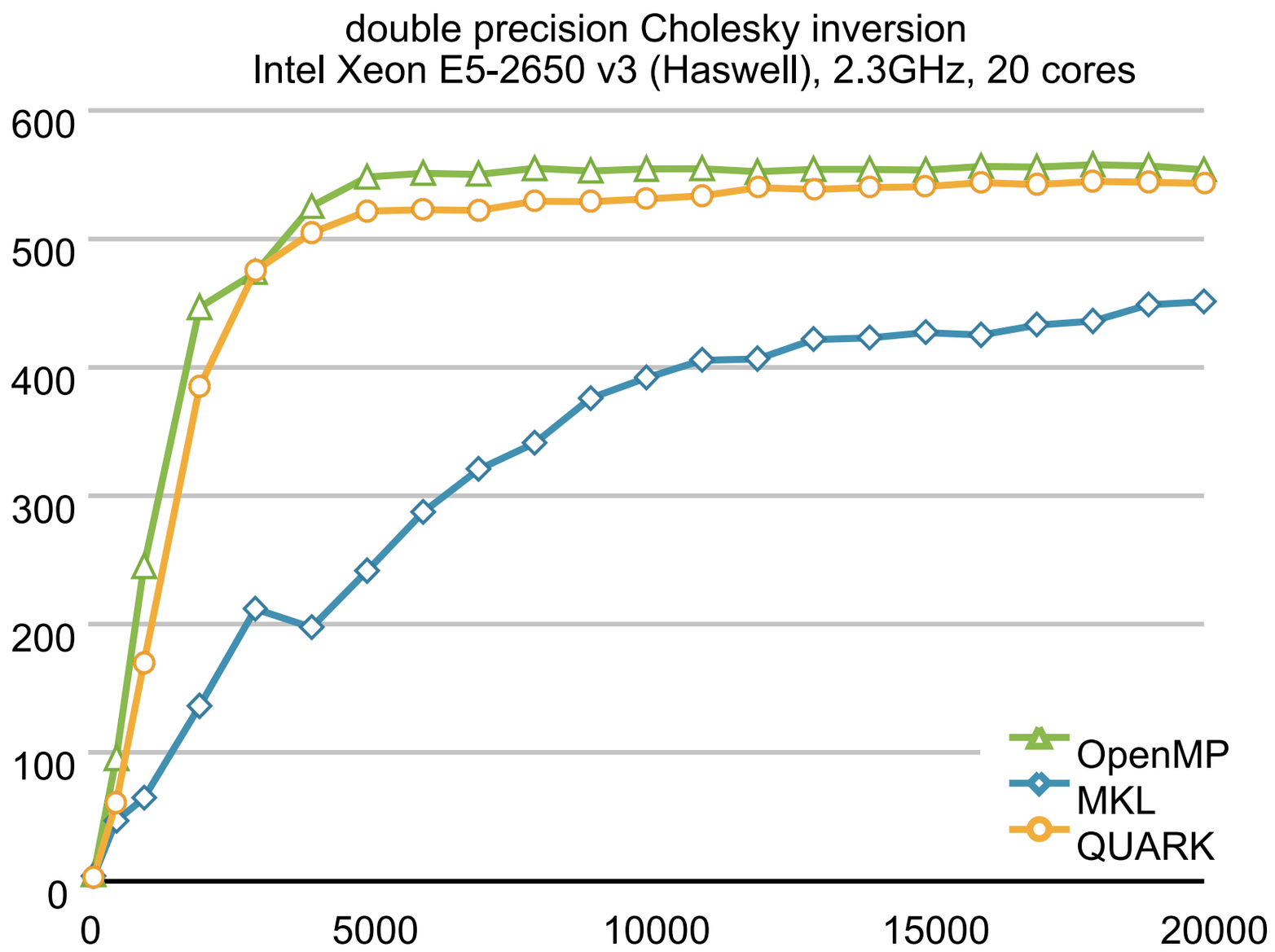
Cholesky matrix inverse



NOTE: Please do not explicitly invert matrices to compute $\boldsymbol{X} = \boldsymbol{A}^{-1}\boldsymbol{B}$.

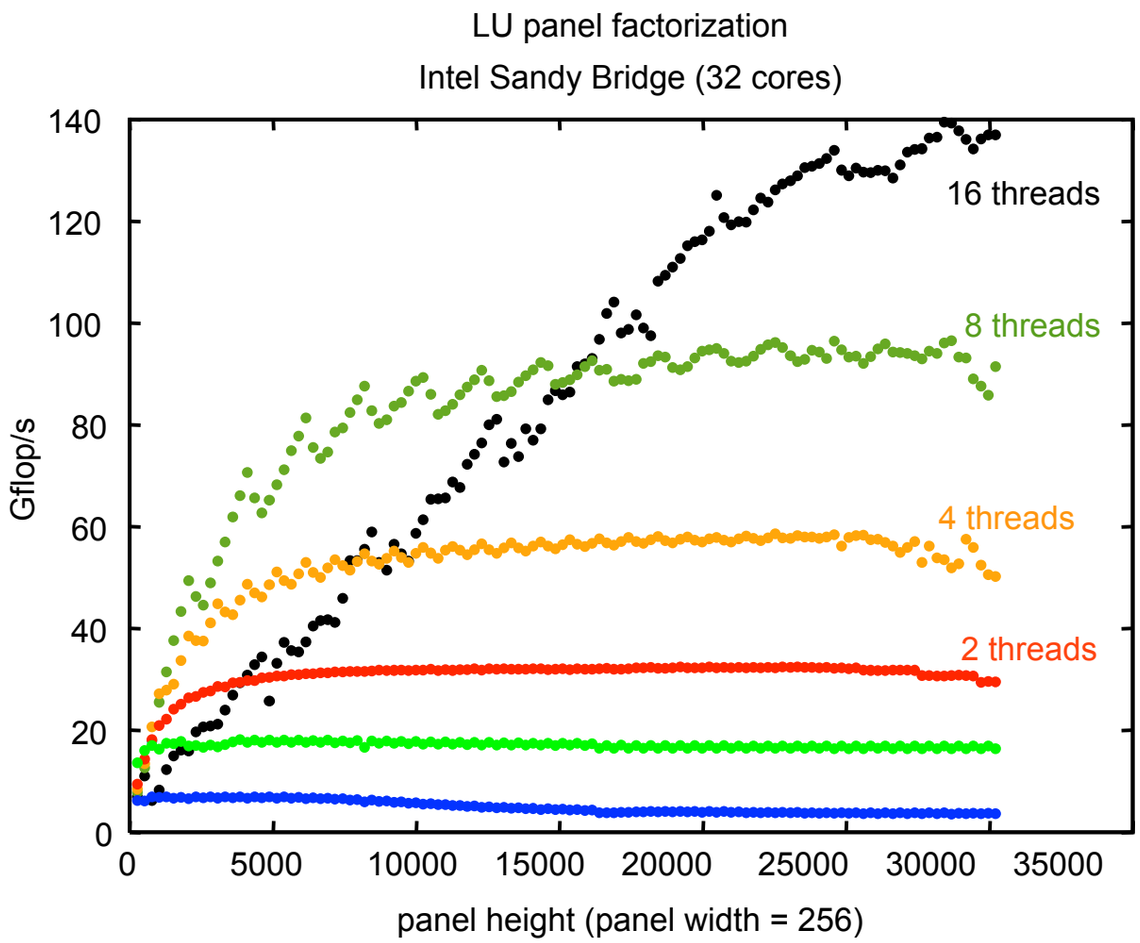
Solve using **gesv**, **posv**, or **sysv**; faster and more accurate than inverting and multiplying

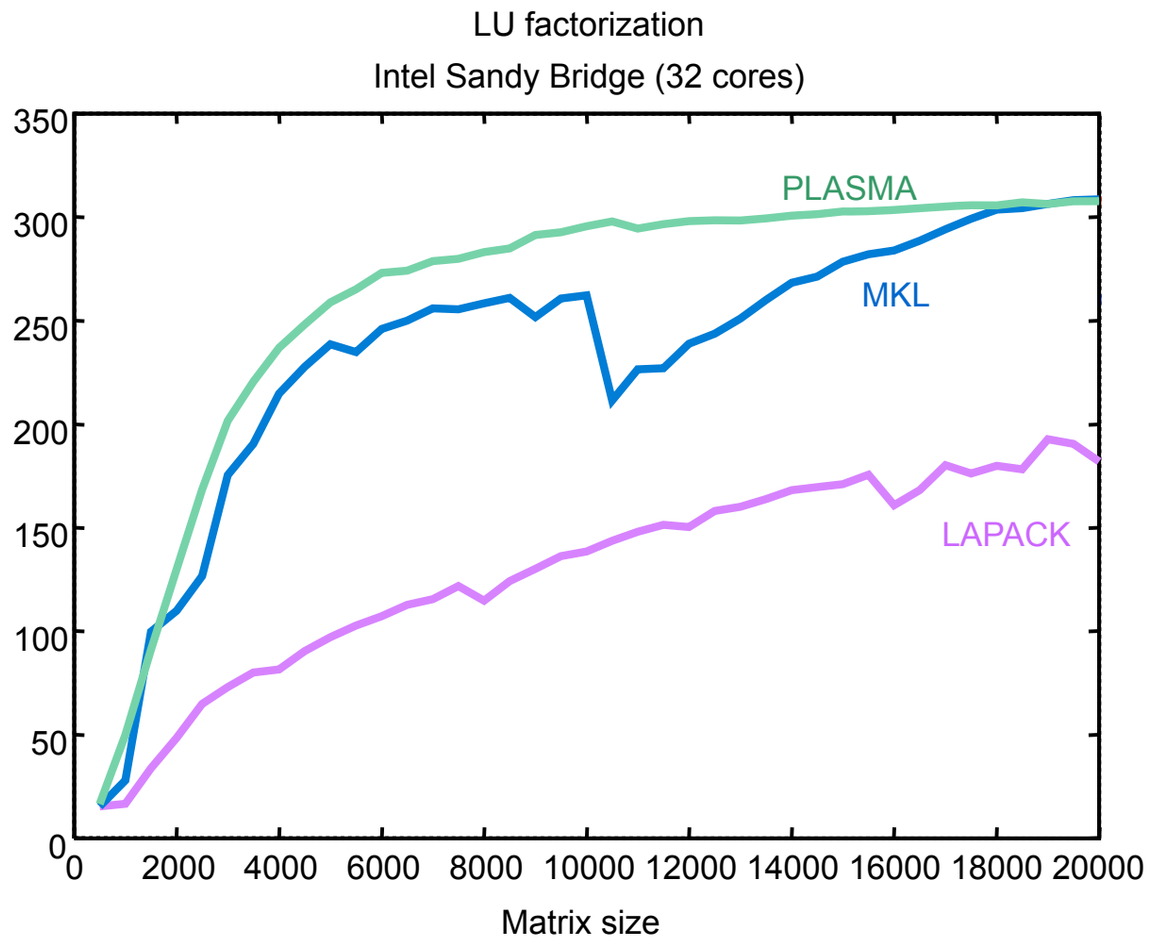
Cholesky inversion performance



LU performance

- Multi-threaded panel
- Factorization reaches peak performance quickly

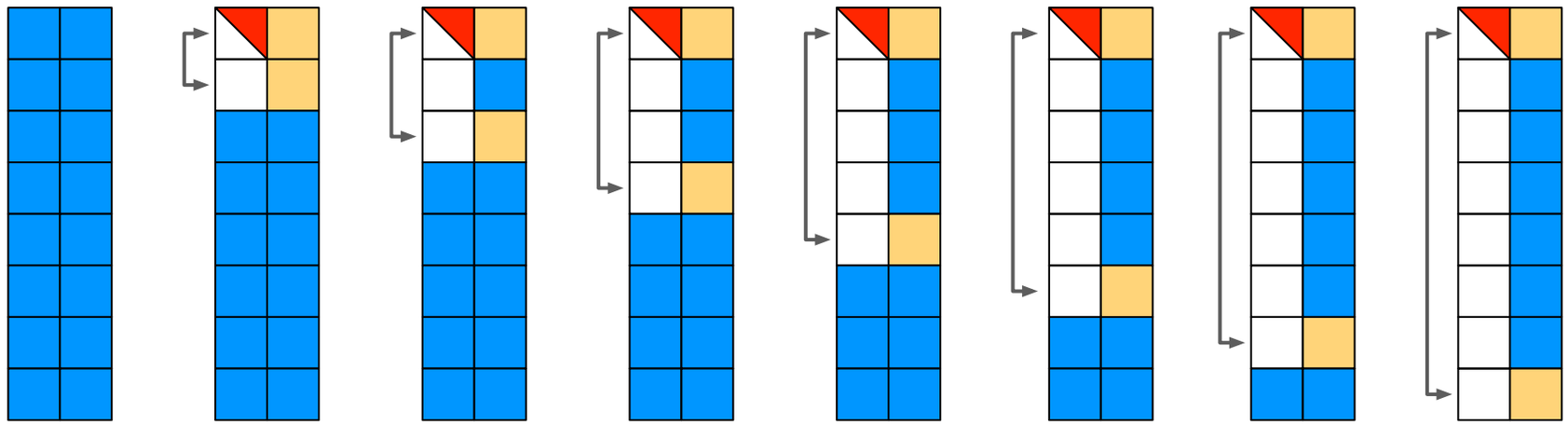




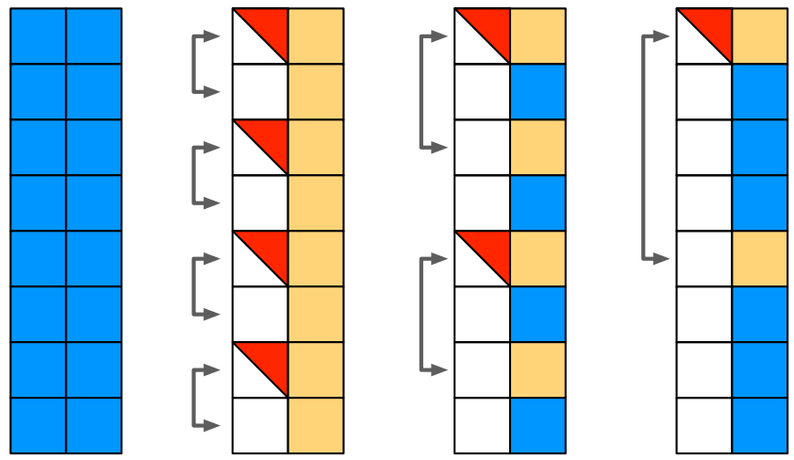
S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, I. Yamazaki, *A survey of recent developments in parallel implementations of Gaussian elimination*, Concurrency and Computation: Practice and Experience, 27(5):1292–1309, 2015. DOI: 10.1002/cpe.3110

PLASMA QR

- Tile QR
  - Great for square matrices
  - Great for multicore
  - Pairwise reductions “domino”

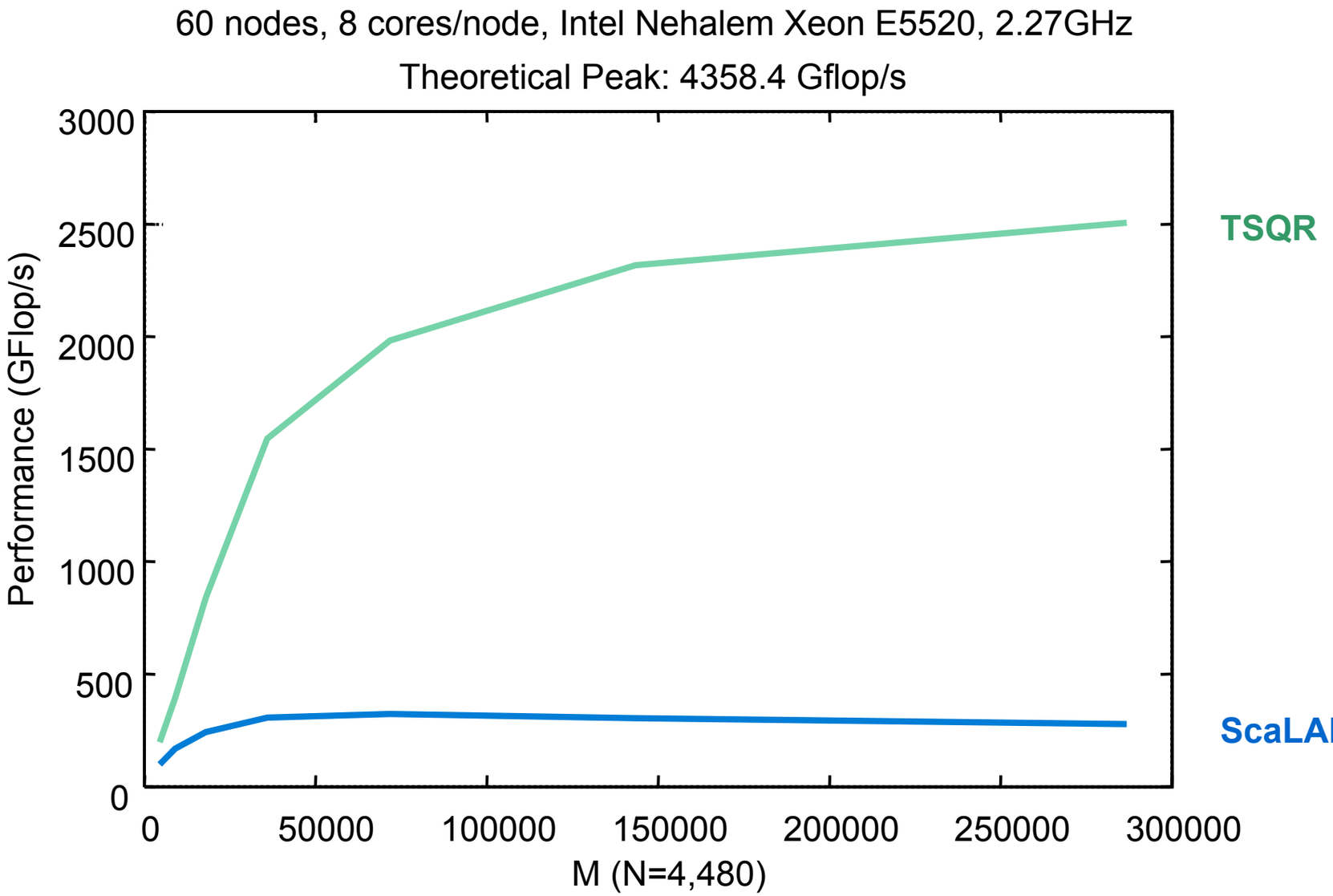


- TSQR / CAQR
  - Tall-skinny QR
  - Communication avoiding QR
  - Tree of pairwise reductions
  - Great for tall-skinny matrices (least squares)
  - Great for distributed memory
  - But triangle-triangle (TT) pairs less efficient than square-square (SS) or triangle-square (TS)



TSQR performance

- Fixed cols, increase rows from square (left) to tall-skinny (right)



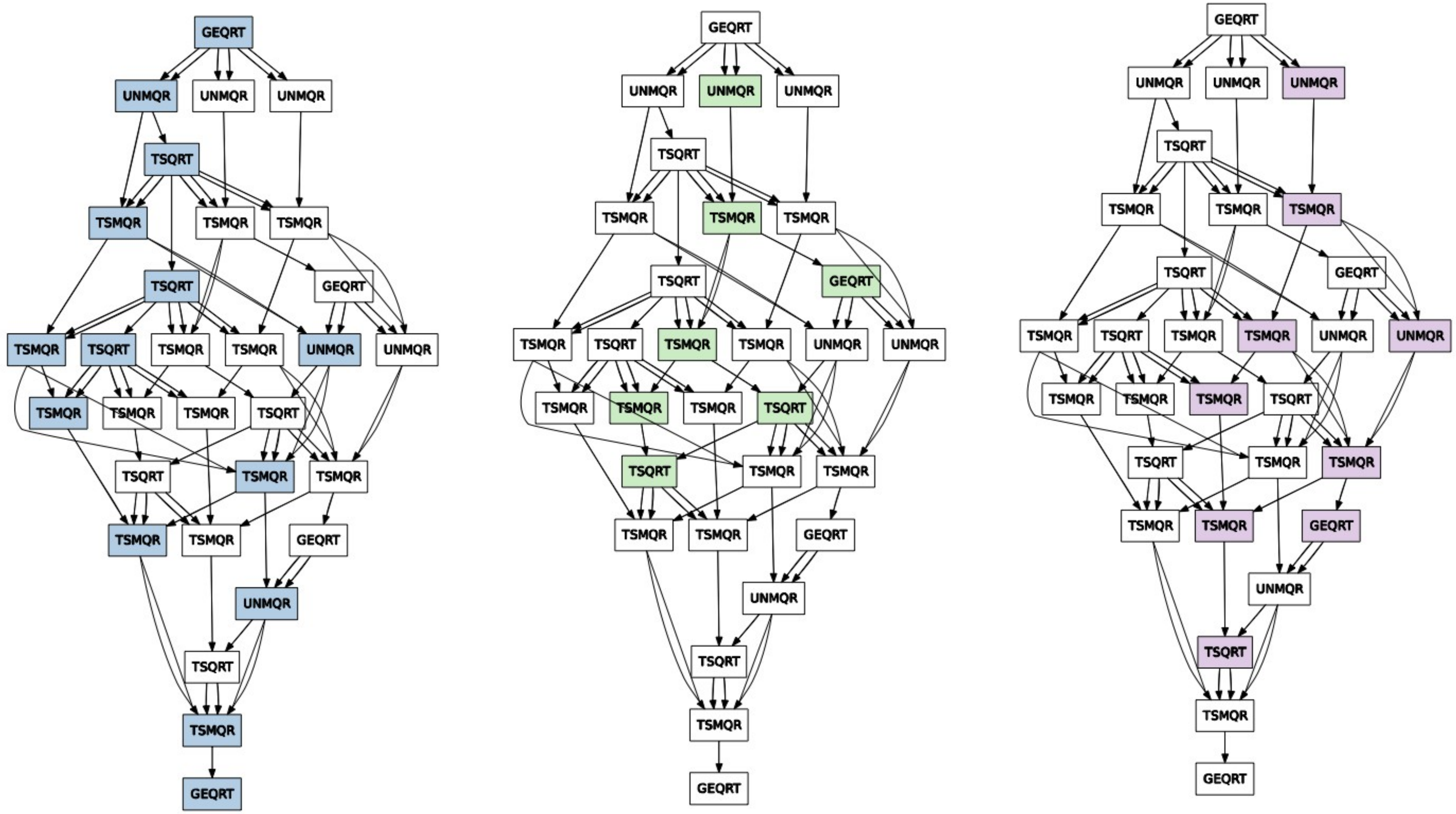
Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

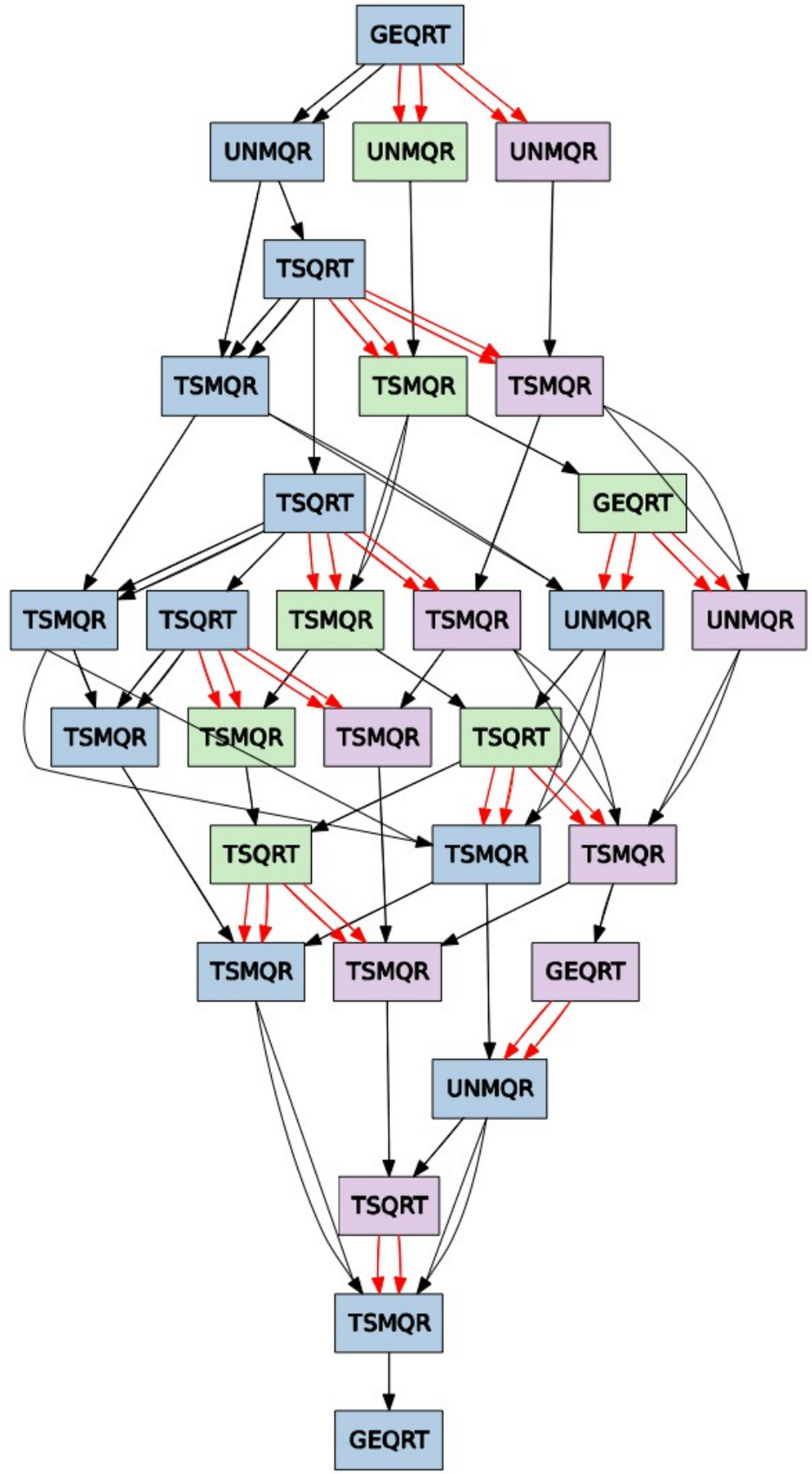
- New Software
  - PLASMA using OpenMP
  - **DPLASMA and PaRSEC**
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

Distributed dataflow execution

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}



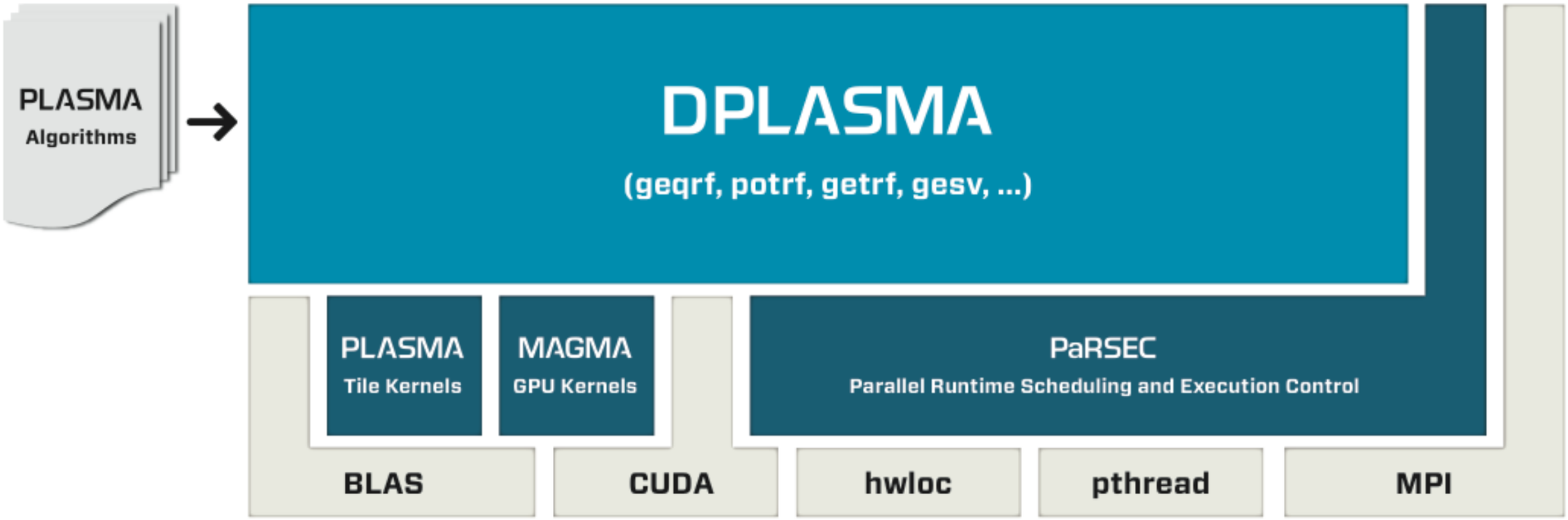
A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}



local depenency

communication

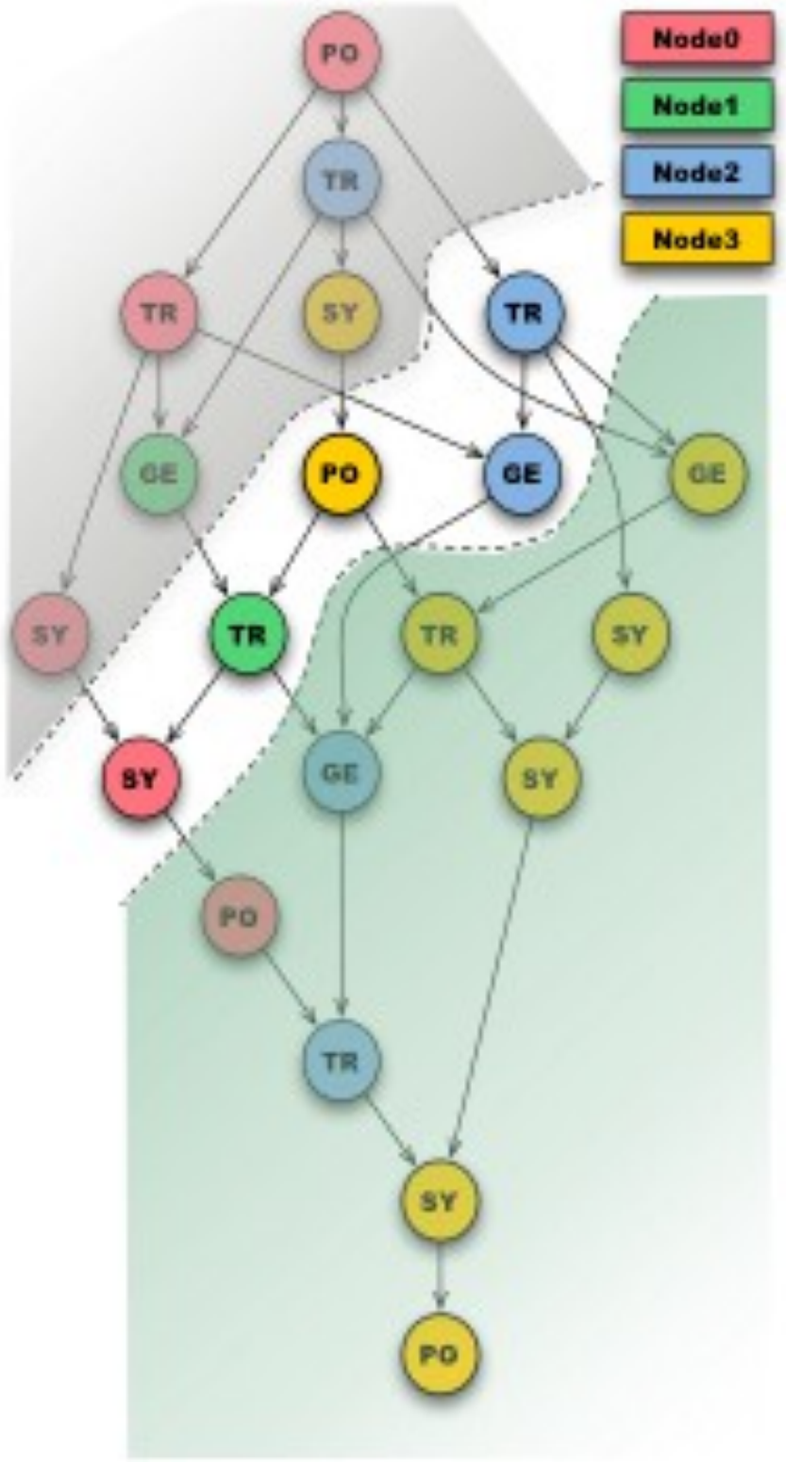
DPLASMA architecture



Parameterized task graph (PTG)

- Symbolic DAG representation
- Problem size independent
- Completely distributed
- DAG is $O(n^3)$,
  PTG avoids explicitly creating it

- Runtime
  - Data-driven execution
  - Locality-aware scheduling
  - Communication overlap



From serial code to PTG

Serial code

```
FOR k=0 TO N-1
  DGEQRT( inout A_kk )
  FOR n=k+1 to N
    DORMQR( in A◹_kk,  inout A_kn )
  FOR m=k+1 to N
    DTSQRT( inout A◹_kk,  inout A_mk )
    FOR n=k+1 to N
      DTSMQR( in A_mk,  inout A_kn,  inout A_mn )
```

PTG representation

DGEQRT$_{kkk}$

1_{ARG} ← $A_{k,k}$ | DTSMQR$_{k,k,k-1}$
1_{ARG} ⇒ DORMQR$_{k,k+1..N,k}$(◹)
1_{ARG} ⇒ DTSQRT$_{k+1,k,k}$(◹)
1_{ARG} ⇒ $A_{k,k}$(◹)

DORMQR$_{knk}$

1_{ARG} ← DGEQRT$_{k,k,k}$(◹)
2_{ARG} ← $A_{k,n}$ | DTSMQR$_{k,n,k-1}$
2_{ARG} ⇒ DTSMQR$_{k+1,n,k}$
2_{ARG} ⇒ $A_{k,n}$

DTSQRT$_{mkk}$

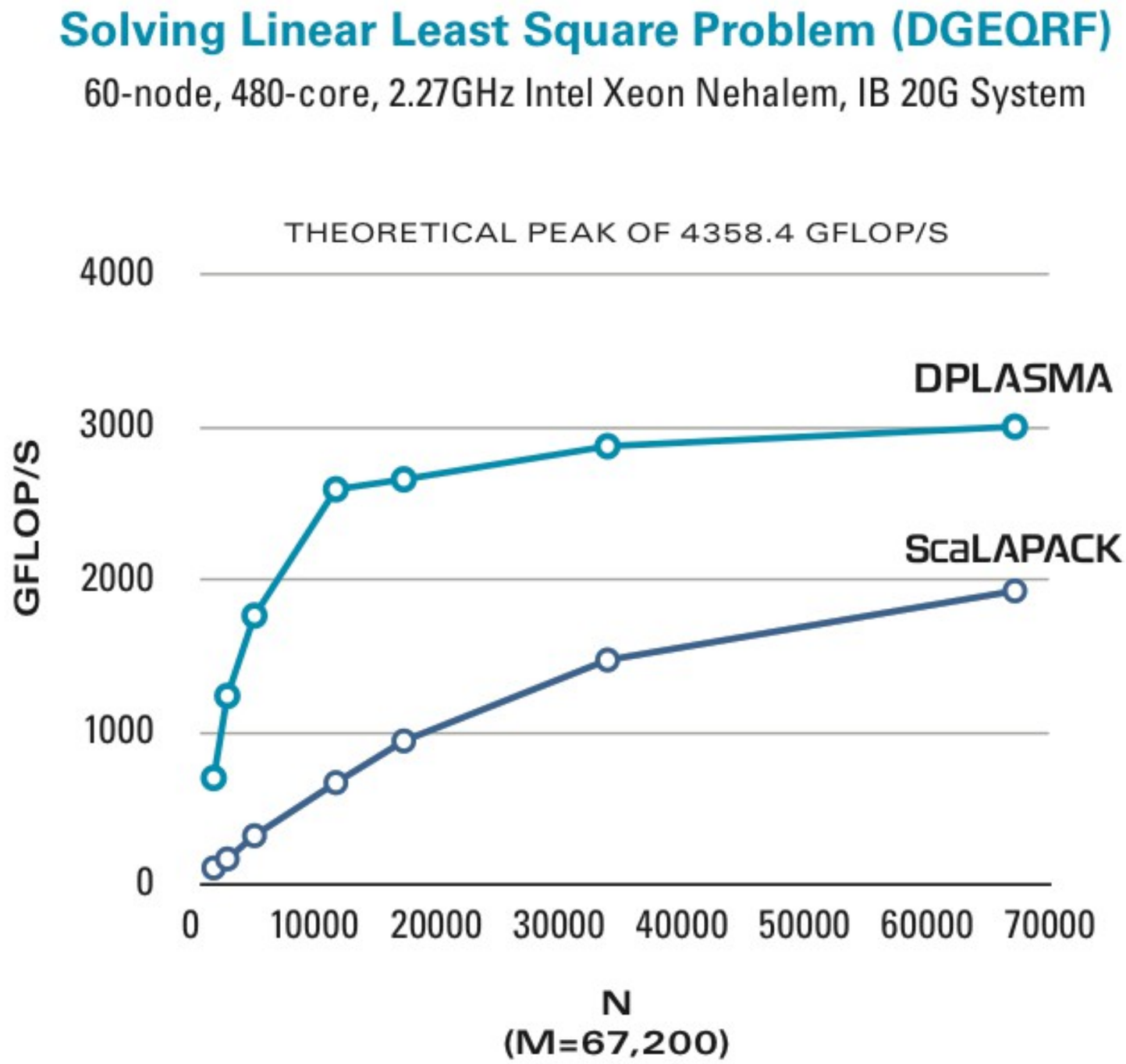
1_{ARG} ← DGEQRT$_{m-1,k,k}$(◹) | DTSQRT$_{m-1,k,k}$(◹)
1_{ARG} ⇒ DTSQRT$_{m+1,k,k}$(◹) | $A_{k,k}$(◹)
2_{ARG} ← $A_{m,k}$ | DTSMQR$_{m,k,k-1}$
2_{ARG} ⇒ DTSMQR$_{m,k+1..N,k}$
2_{ARG} ⇒ $A_{m,k}$

DTSMQR$_{mnk}$

1_{ARG} ← DTSQRT$_{m,k,k}$
2_{ARG} ← DORMQR$_{m-1,n,k}$ | DTSMQR$_{m-1,n,k}$
2_{ARG} ⇒ DTSMQR$_{m+1,n,k}$ | $A_{n,k}$
3_{ARG} ← $A_{m,n}$ | DTSMQR$_{m,n,k-1}$
3_{ARG} ⇒ DGEQRT$_{m,n,k+1}$ | DORMQR$_{m,n,k+1}$ |
⇒ DTSQRT$_{m,n,k+1}$ | DTSMQR$_{m,n,k+1}$ |
⇒ $A_{m,n}$

QR performance

- Fixed rows, increase cols, from tall-skinny (left) to square (right)

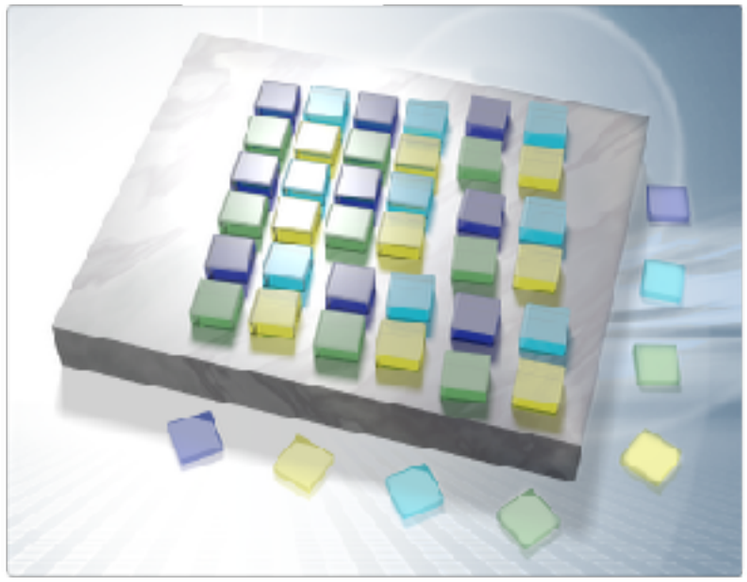


Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - **SLATE**
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - Case study: 2-stage SVD

SLATE



- C++11, MPI, OpenMP, cuBLAS
- Map of tiles
  - A.tiles_[{ i, j, device }]
    where { i, j, device } is std::tuple< int, int, int >,
    i and j are always global tile indices
  - Supports both contiguous (PLASMA) and non-contiguous (ScaLAPACK) tiles
  - Tiles can be allocated independently, unlike PLASMA
- Trailing matrix updates
  - CPU: OpenMP tasks or Batched BLAS
  - GPU: Batched BLAS
    - Batched BLAS is many small, independent gemms in one call, in parallel
- Explicit look-ahead
  - Factor next panel(s) while trailing matrix is updating
  - Hides communication

SLATE components

- BLAS++
- LAPACK++
  - C++ wrappers around BLAS and LAPACK routines
  - Enables C++ template code
  - Removes 1st letter for precision — determined by arguments
    - gemm(A, B, C) ⇒ sgemm, dgemm, cgemm, or zgemm

      depending on type of A, B, and C.
  - Names and arguments consistent across all 4 precisions
    - herk of real matrix ⇒ syrk
    - geev always takes complex vector of eigenvalues (w), instead of split-complex as in LAPACK's dgeev (wr, wi)
  - Allocates optimal workspace internally
  - Throws C++ exceptions on hard errors (invalid dimensions, ...); returns numerical errors (singular matrix, ...)

Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - **MAGMA for CUDA, OpenCL, or Xeon Phi**
  - Case study: 2-stage SVD

Challenges of GPUs

- High levels of parallelism
- Expensive branches (if-then-else)
- Computation vs. communication gap growing
  - NVIDIA Pascal P100 has 4670 Gflop/s, 732 GB/s memory, 16 GB/s PCIe (80 GB/s NVLINK)
- Use hybrid approach
  - Small, non-parallelizable tasks on CPU
  - Large, parallel tasks on GPU
  - Overlap communication & computation

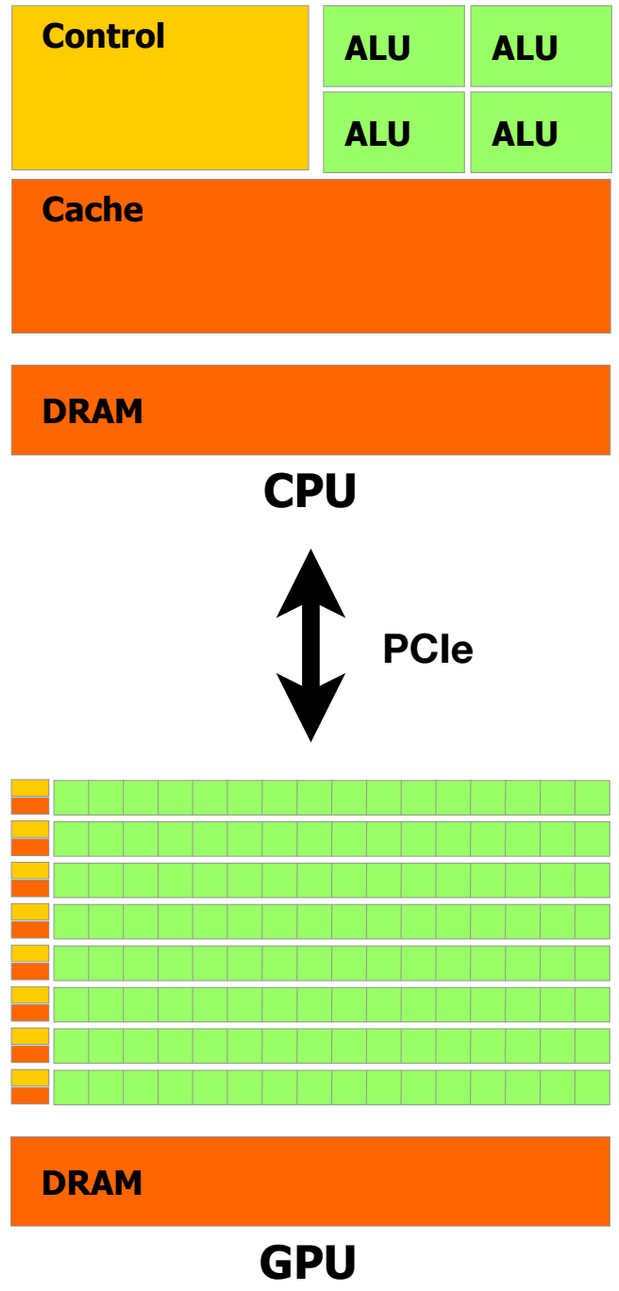


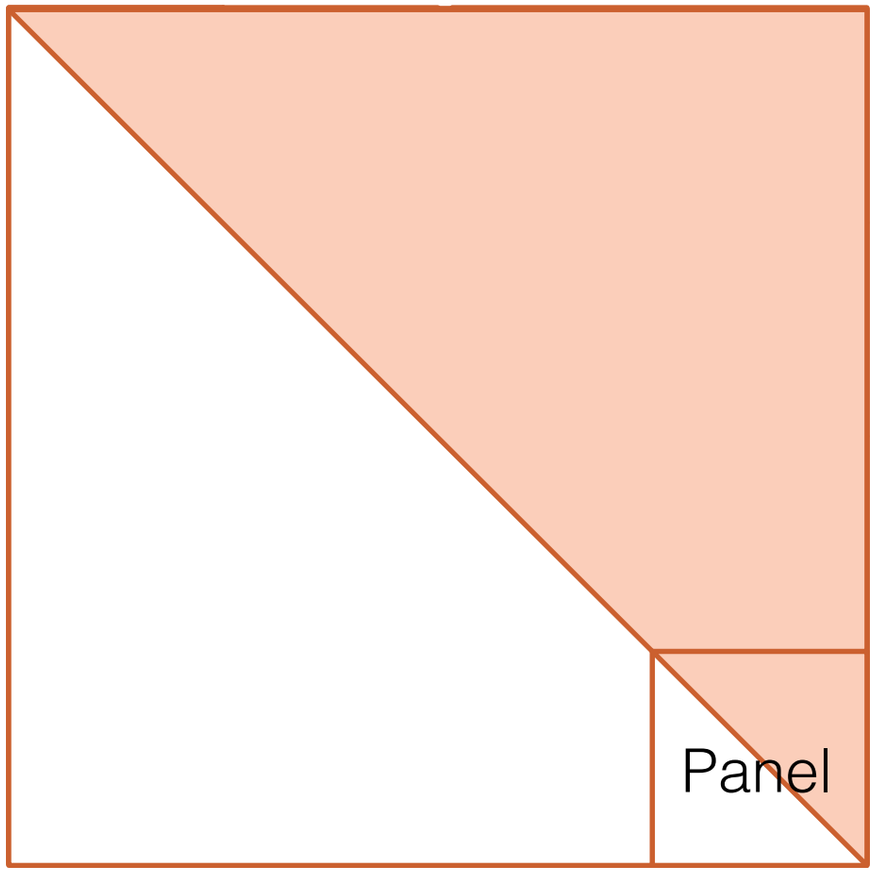
Figure from NVIDIA CUDA C Programming Guide

One-sided factorization

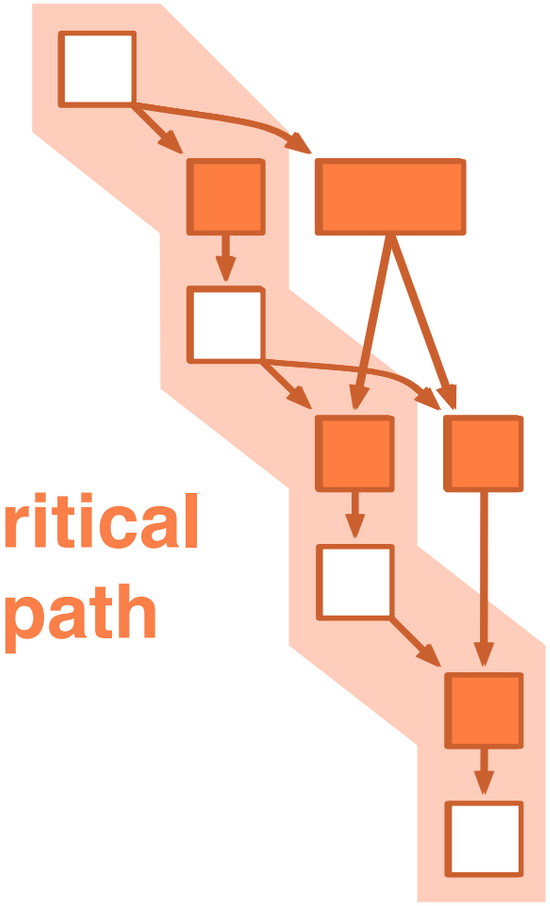
- LU, Cholesky, QR factorizations for solving linear systems

Level 2 BLAS on CPU

Level 3 BLAS on GPU

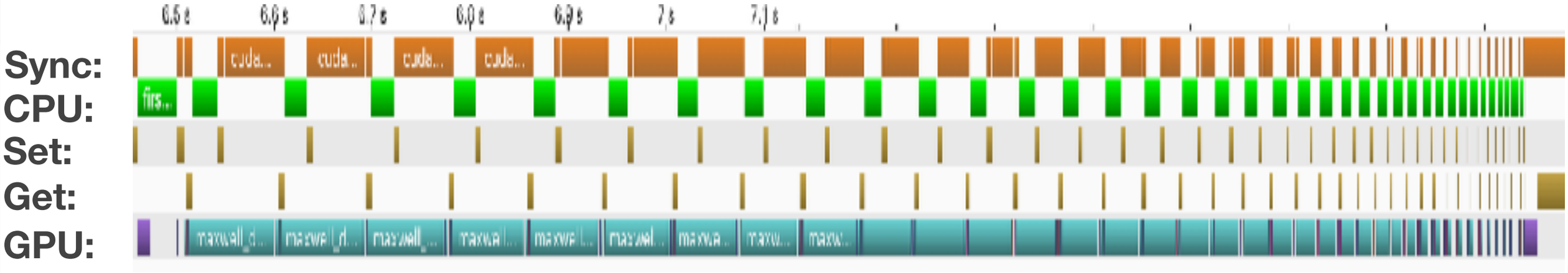


DAG



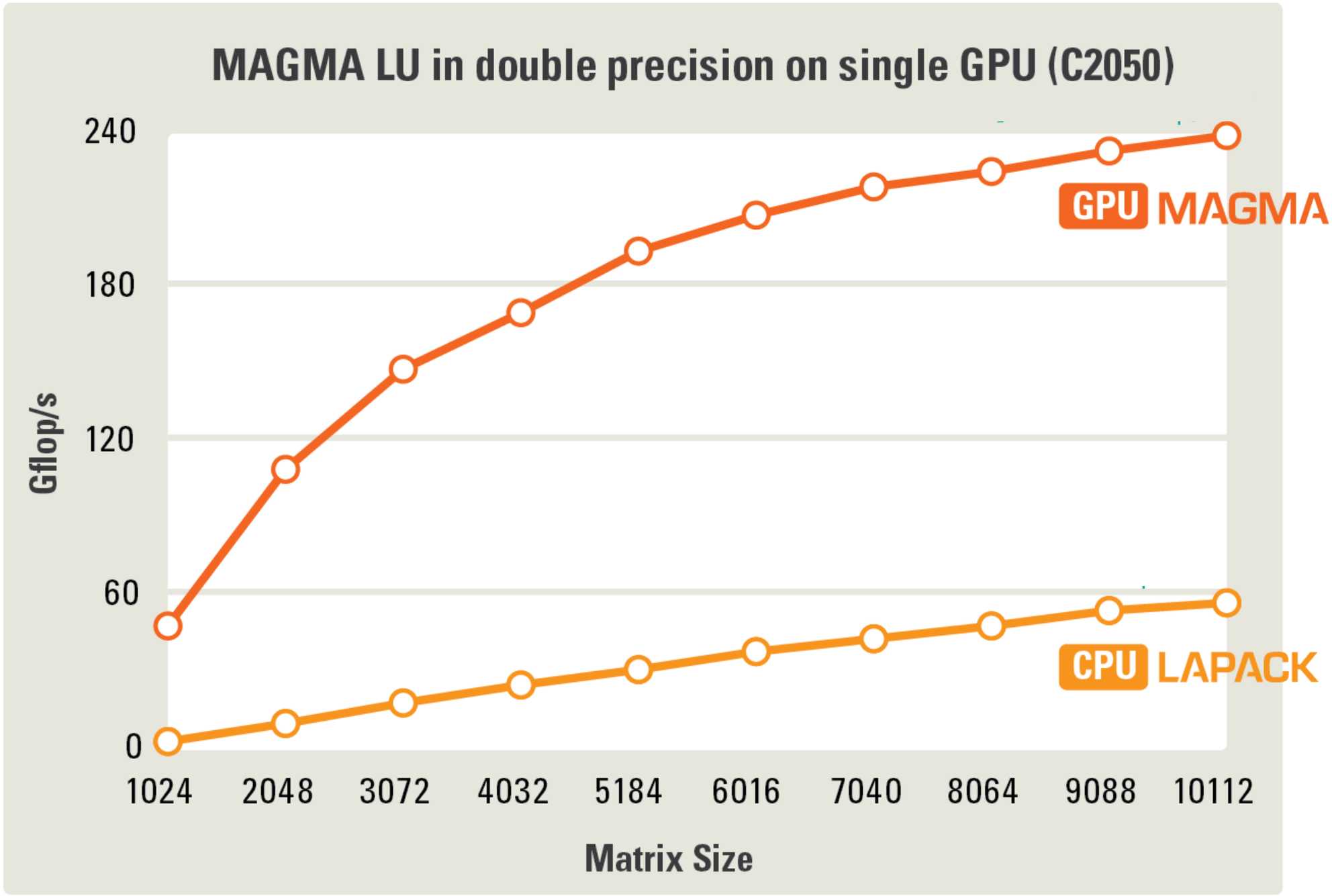
Execution trace

- **Panels** on CPU **(green)** and **set/get communication (brown)** overlapped with **trailing matrix updates (teal)** on GPU
- Goal to keep GPU busy all the time; CPU may idle



LU factorization (dgetrf), n = 20000
P100 GPU, 2 × 10-core 2.3 GHz Haswell

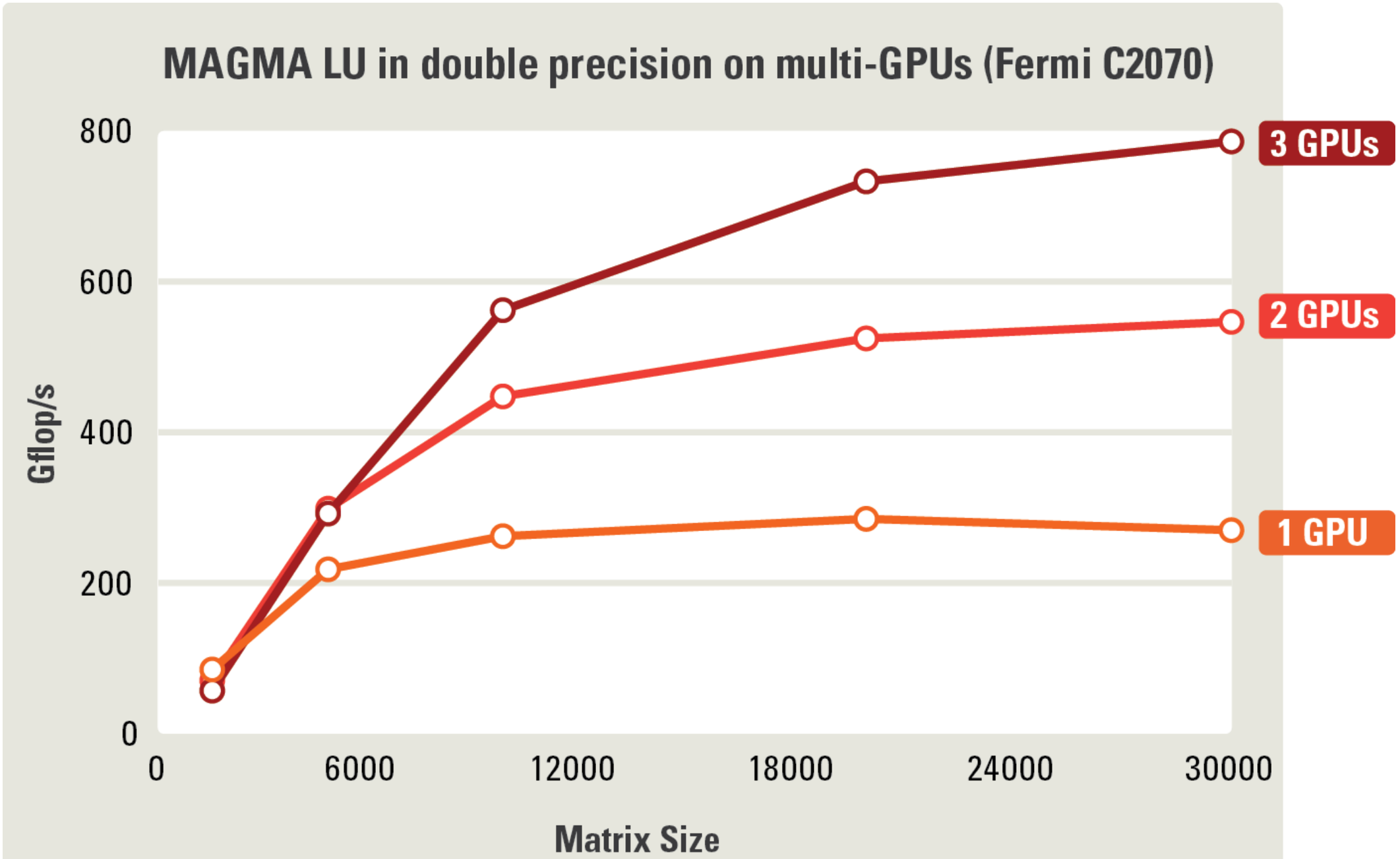
- Optimization: for LU, we transpose matrix on GPU so row-swaps are fast



GPU Fermi C2050 (448 CUDA Cores @ 1.15 GHz)
+ Intel Q9300 (4 cores @ 2.50 GHz)
DP peak **515** + **40** GFlop/s
Power * ~**220** W

CPU AMD Istanbul
[8 sockets x 6 cores (48 cores) @2.8GHz]
DP peak **538** GFlop/s
Power * ~**1,022** W

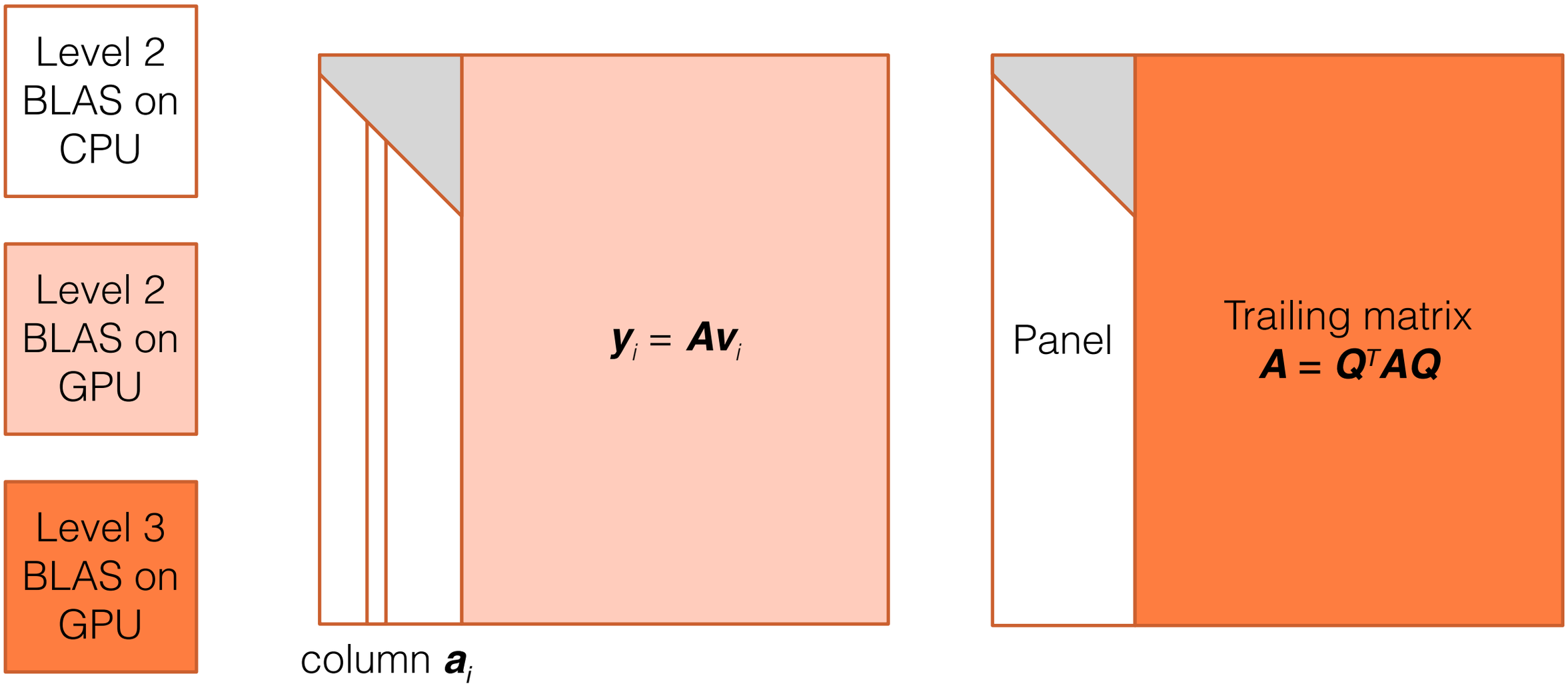
* Computation consumed power rate (total system rate minus idle rate), measured with KILL A WATT PS, Model P430



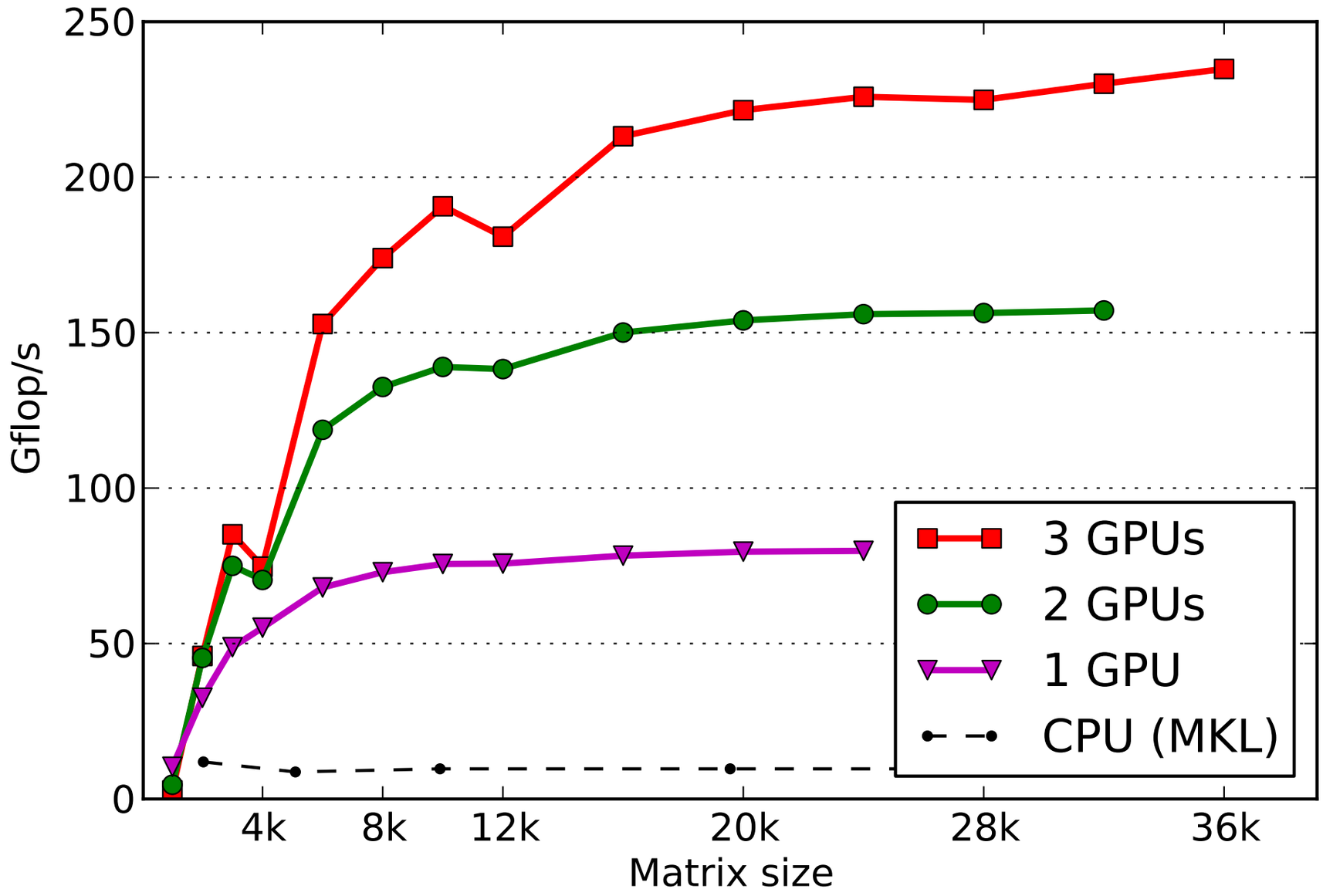
Keeneland system, using one node
3 Nvidia GPUs (M2070 @ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

Two-sided factorization

- Hessenberg, tridiagonal factorizations for eigenvalue problems



MAGMA Hessenberg in double precision



Keeneland system, using one node

3 Nvidia GPUs (M2070 @ 1.1 GHz, 5.4 GB)

2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

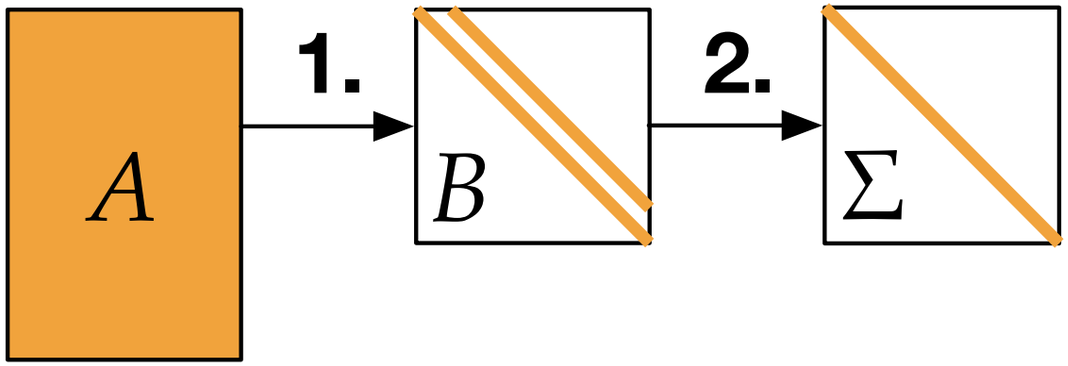
Outline

- Legacy Software
  - BLAS
  - LINPACK
  - LAPACK
  - ScaLAPACK

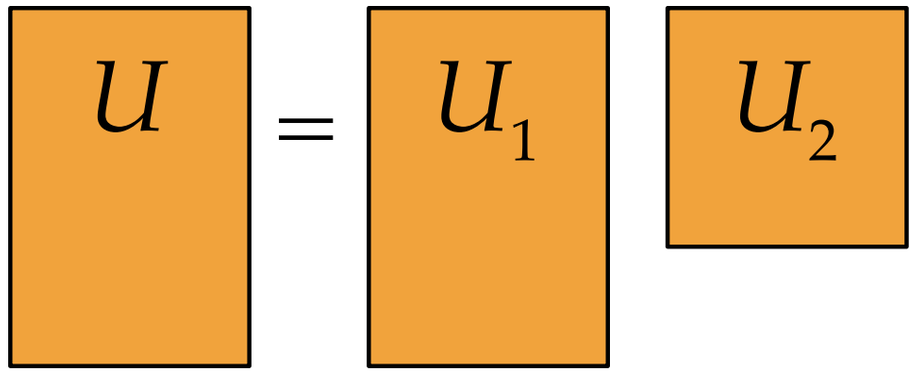
- New Software
  - PLASMA using OpenMP
  - DPLASMA and PaRSEC
  - SLATE
  - SLATE
  - MAGMA for CUDA, OpenCL, or Xeon Phi
  - **Case study: 2-stage SVD**

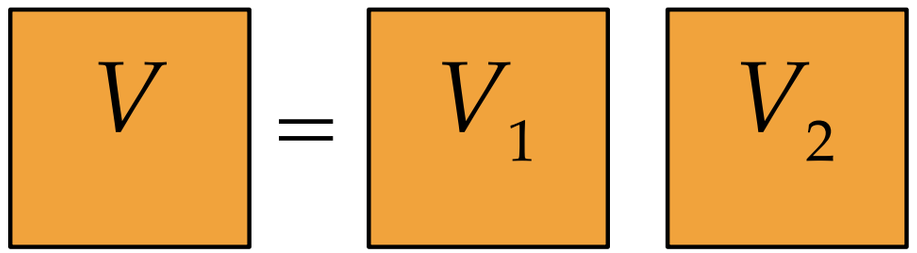
Computing SVD in 3 Phases

$A = U_1 B V_1^T$ $B = U_2 \Sigma V_2^T$



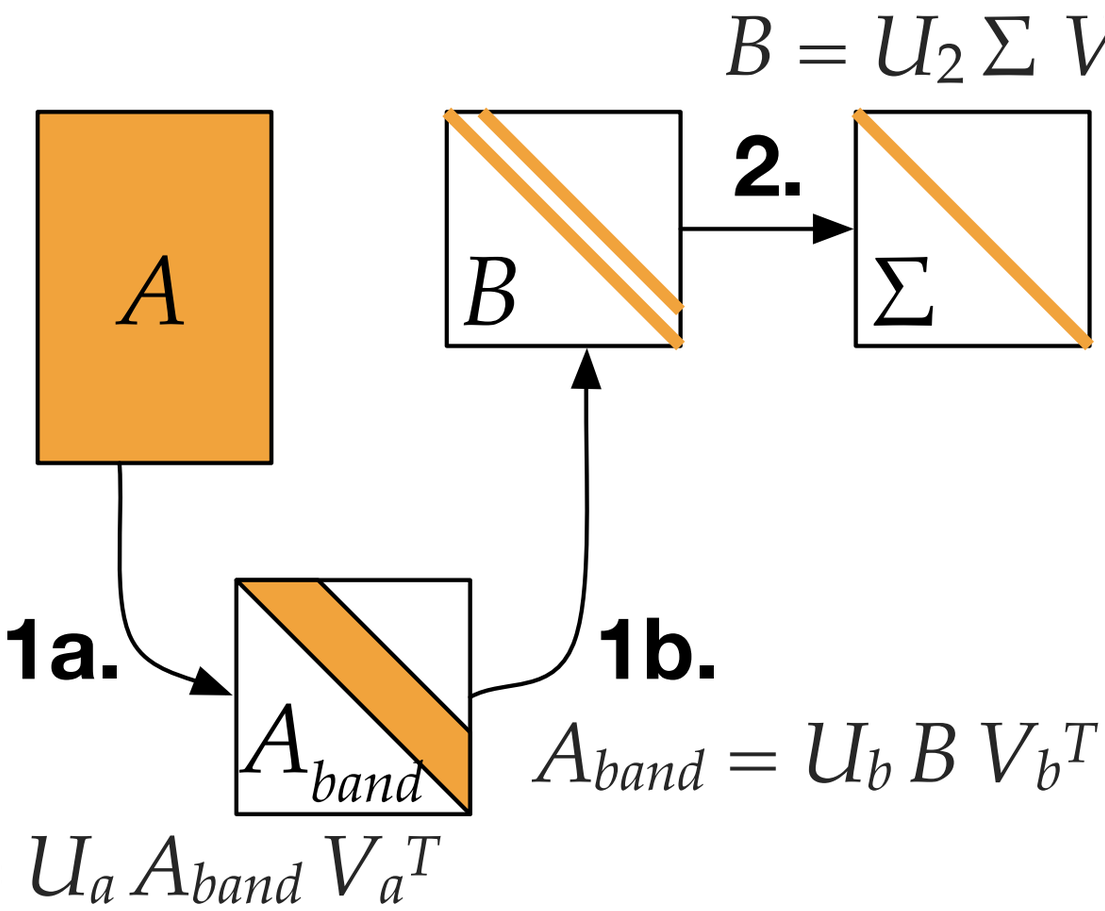
3.





1. Reduction to bidiagonal form
2. Bidiagonal SVD
3. Back transform singular vectors

Computing SVD in 3 Phases



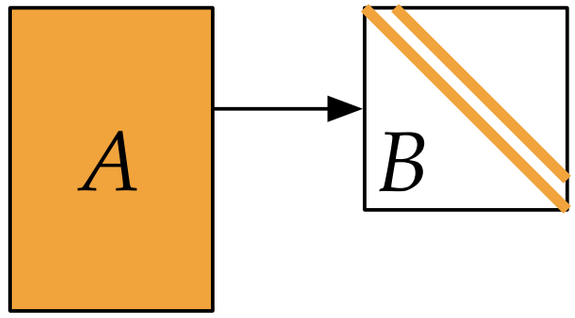
3.

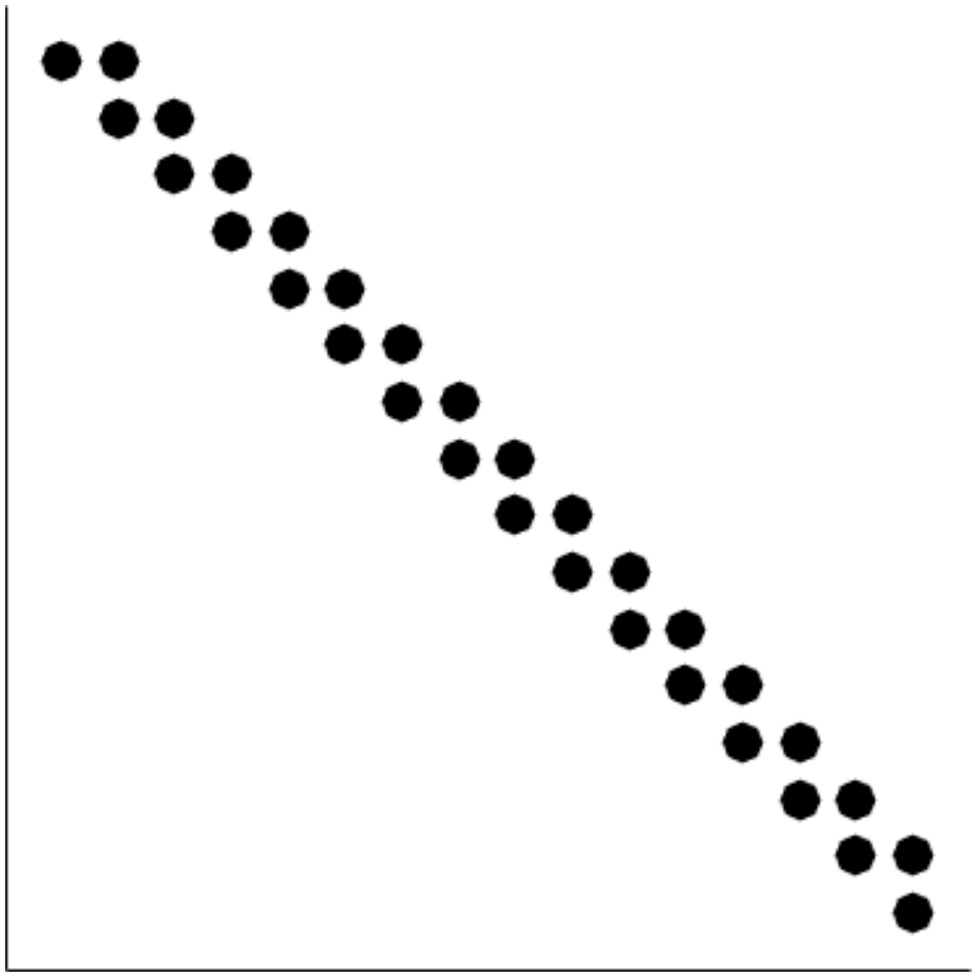
$$U = U_a \; U_b \; U_2$$

$$V = V_a \; V_b \; V_2$$

1. Reduction to bidiagonal form (2 stage)
   - 1a. Full to band
   - 1b. Band to bidiagonal
2. Bidiagonal SVD
3. Back transform singular vectors

One stage reduction

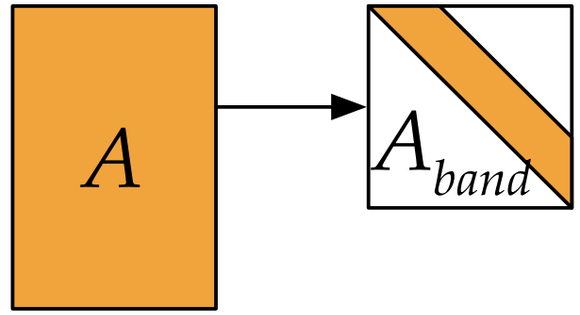


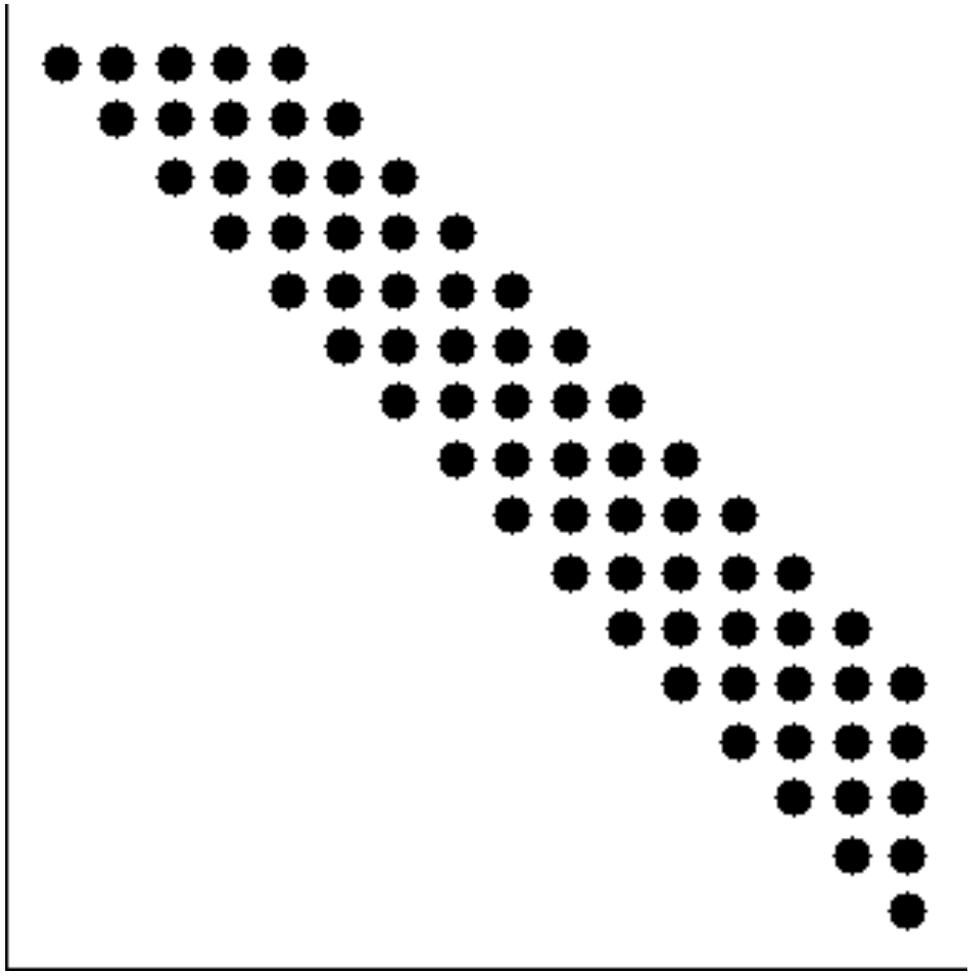


- $\frac{4}{3}n^3$ flops in BLAS 2 gemv
- $\frac{4}{3}n^3$ flops in BLAS 3 gemm
- Performance limited to twice gemv speed

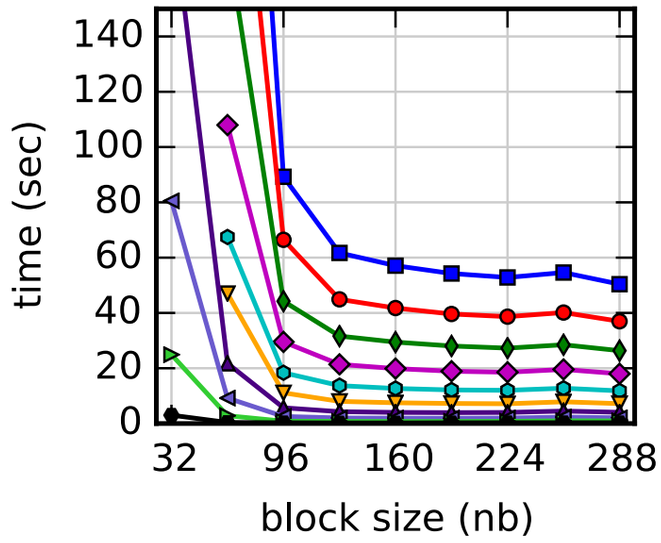
- For $i = 1$ to n by n_b
  - Eliminate cols $i : i + n_b$ and rows $i : i + n_b$
  - Update trailing matrix using block Householder (BLAS 3 matrix multiplies)

Two stage reduction



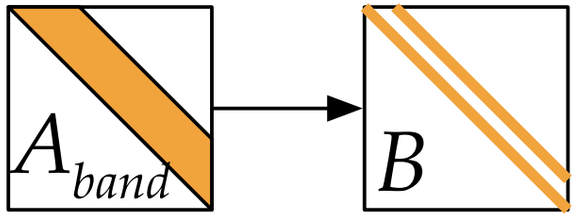


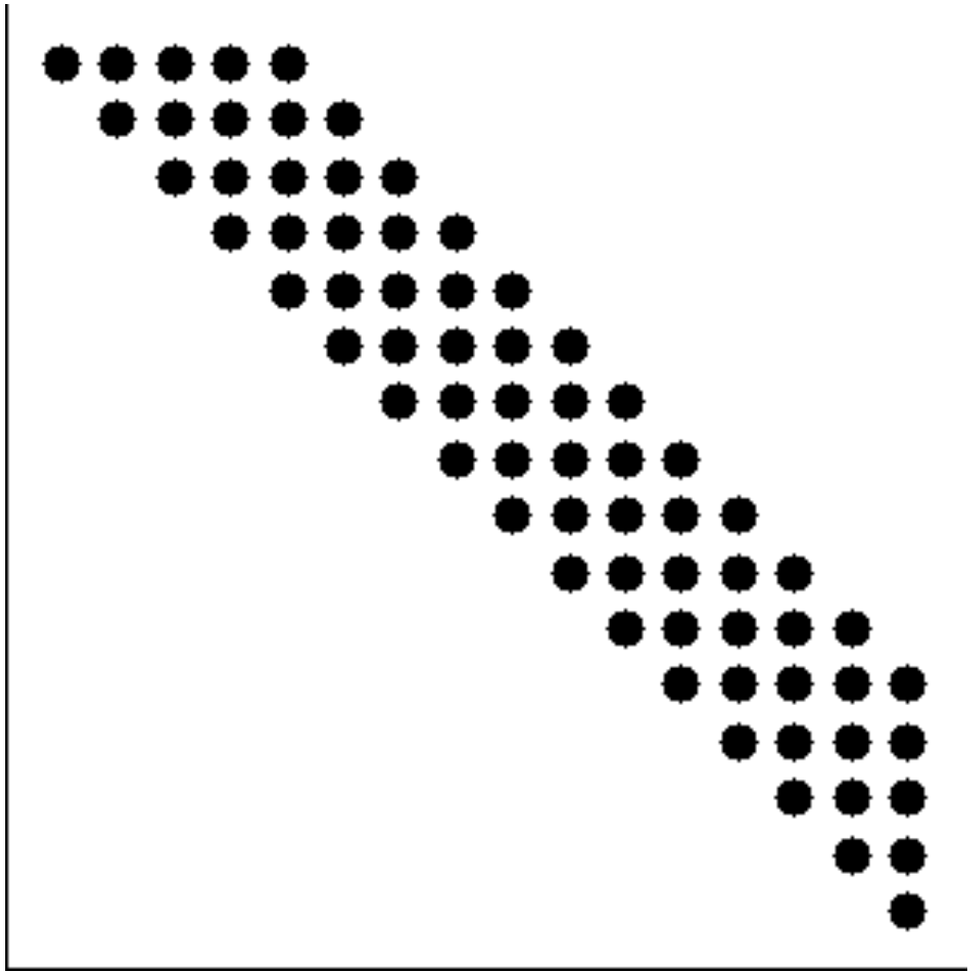
- $\frac{8}{3}n^3$ flops in BLAS 3 gemm
- More efficient with **large** bandwidth n_b



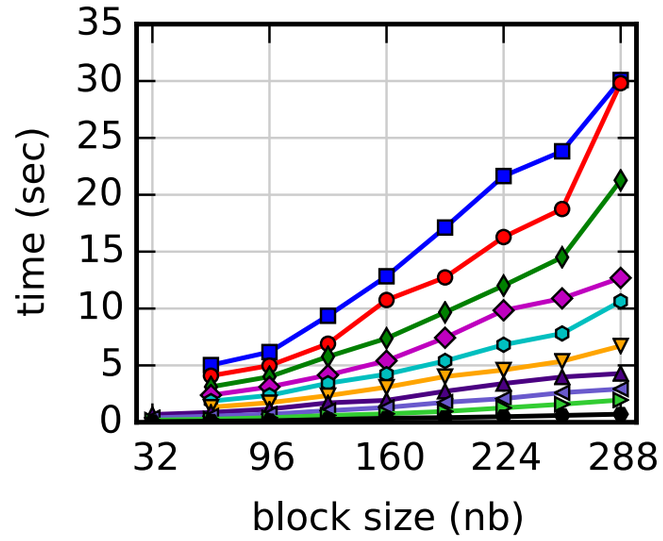
- **1st stage: full to band**
  - QR factorize panel, update trailing matrix
  - LQ factorize panel, update trailing matrix

Two stage reduction





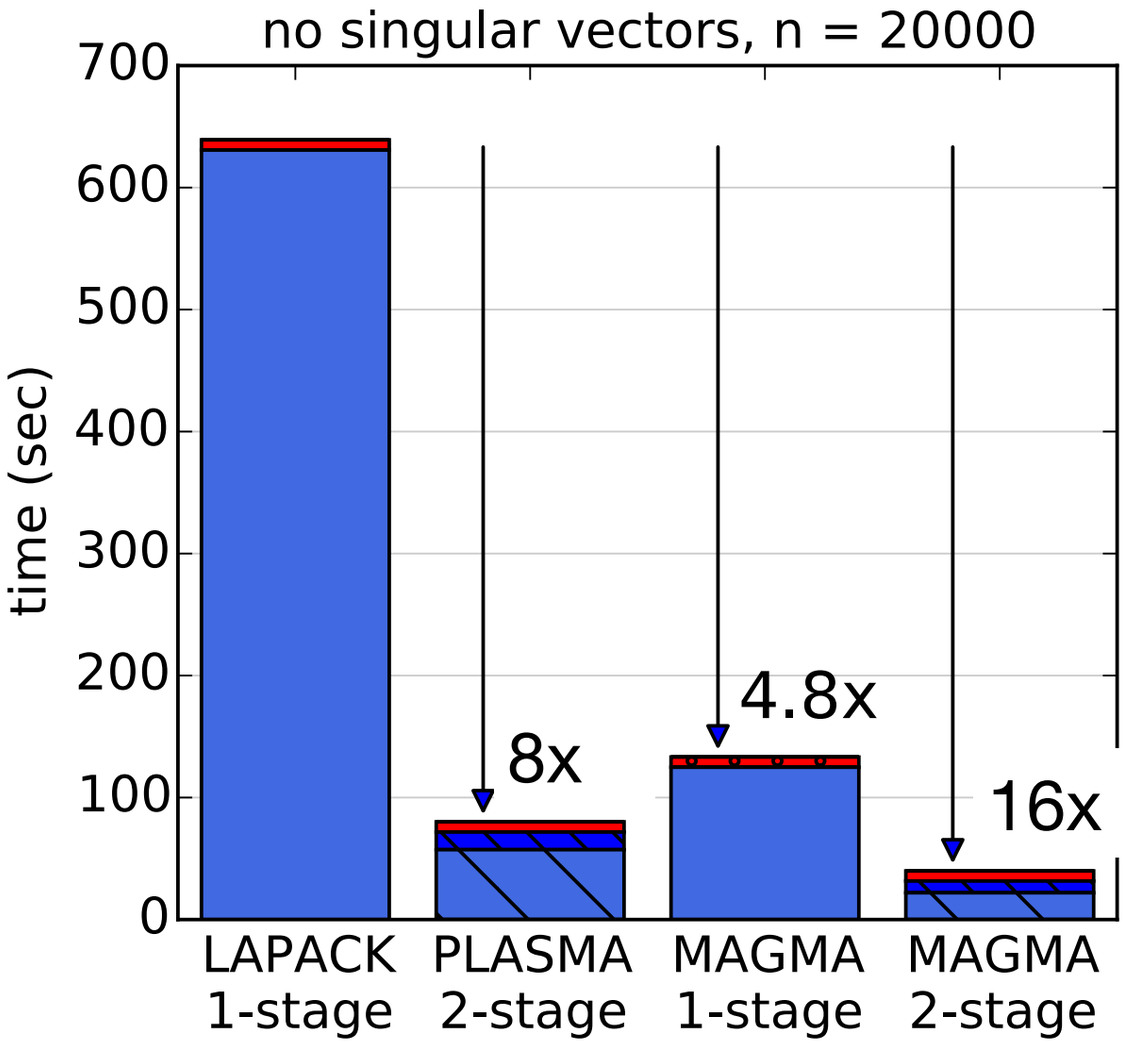
- $O(n_b\, n^2)$ flops in BLAS 2
- More efficient with **small** bandwidth n_b
- Tune optimal n_b

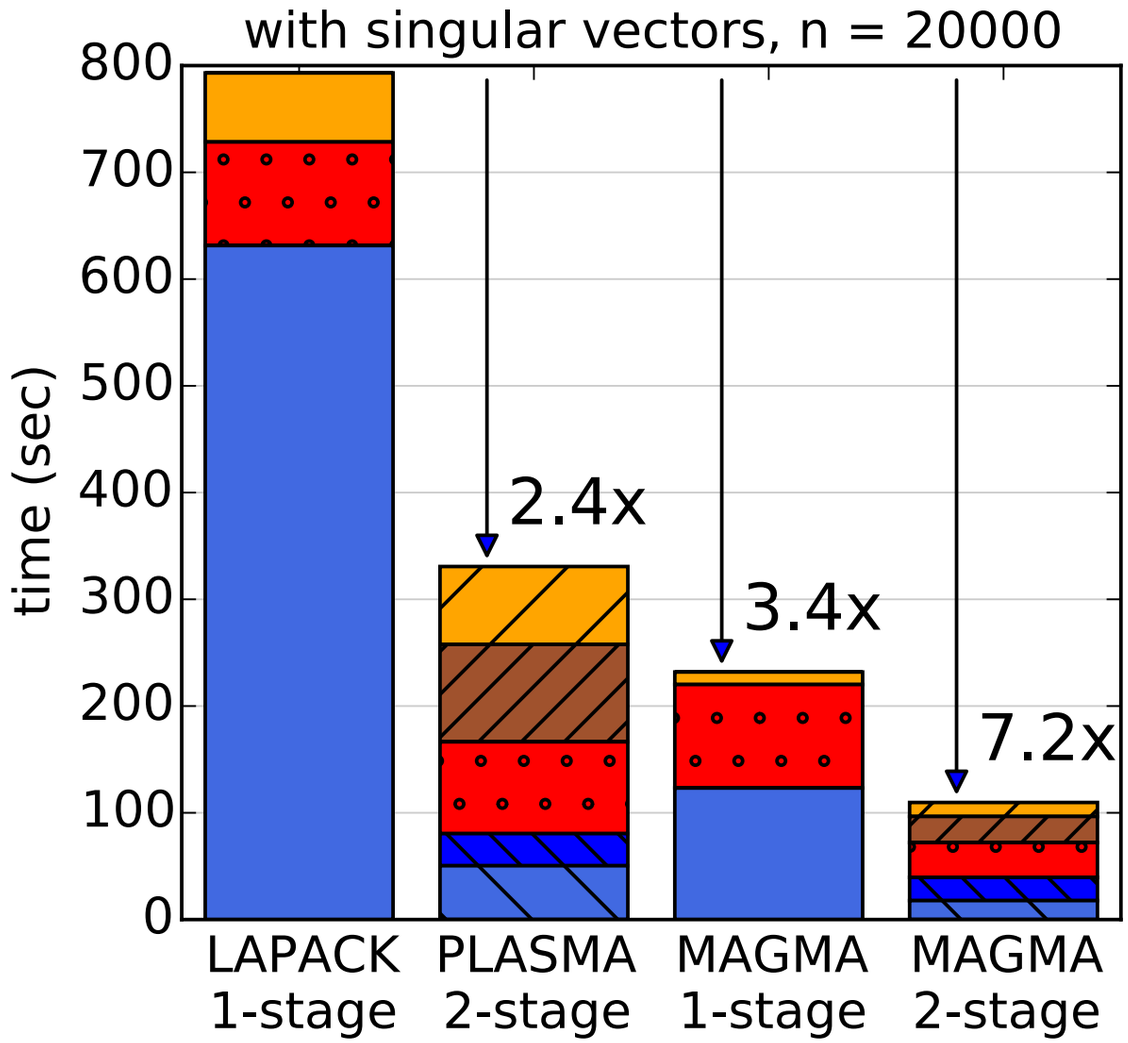


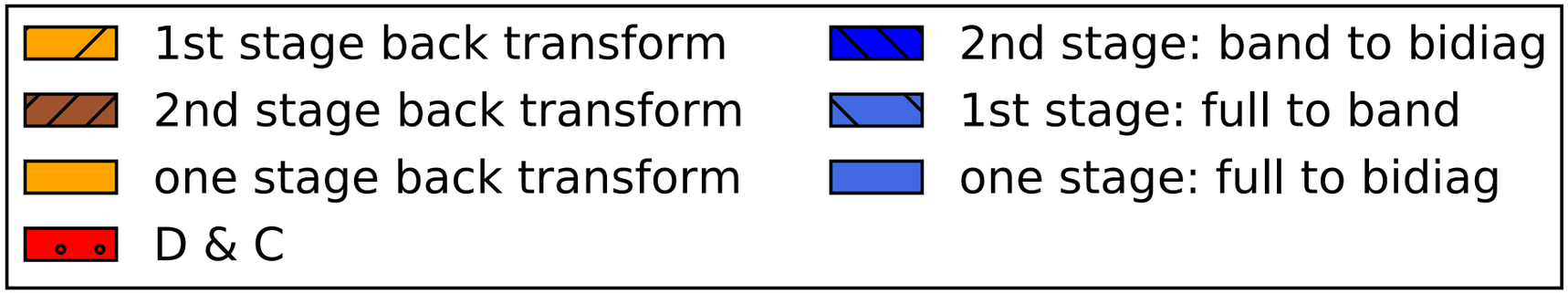
- **2nd stage: band to bidiagonal**
  - Cache friendly kernels
  - Pipeline multiple sweeps in parallel

Accelerated phases

lower is better







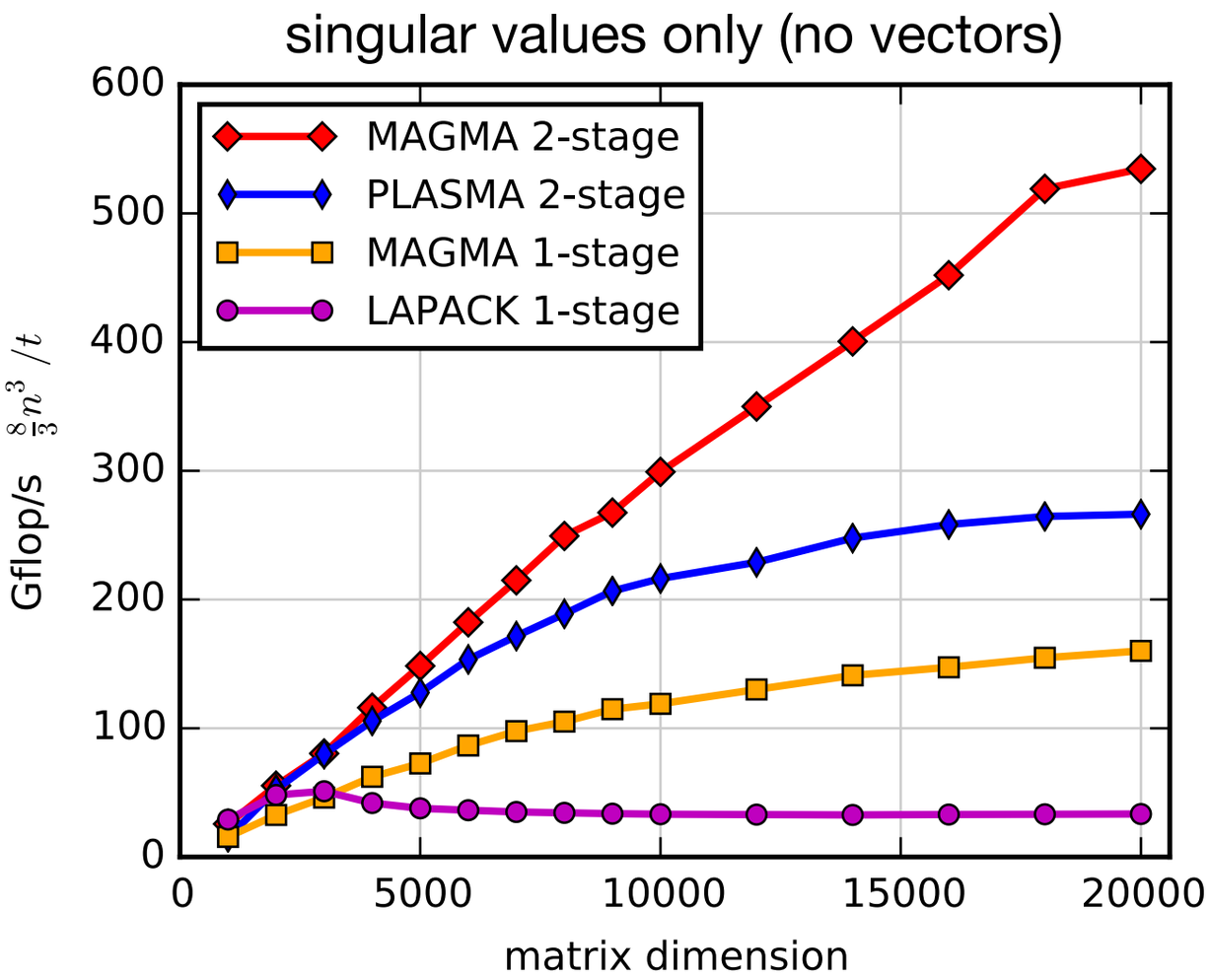
NVIDIA Pascal P100
Intel Haswell 2x10 core 2.3 GHz

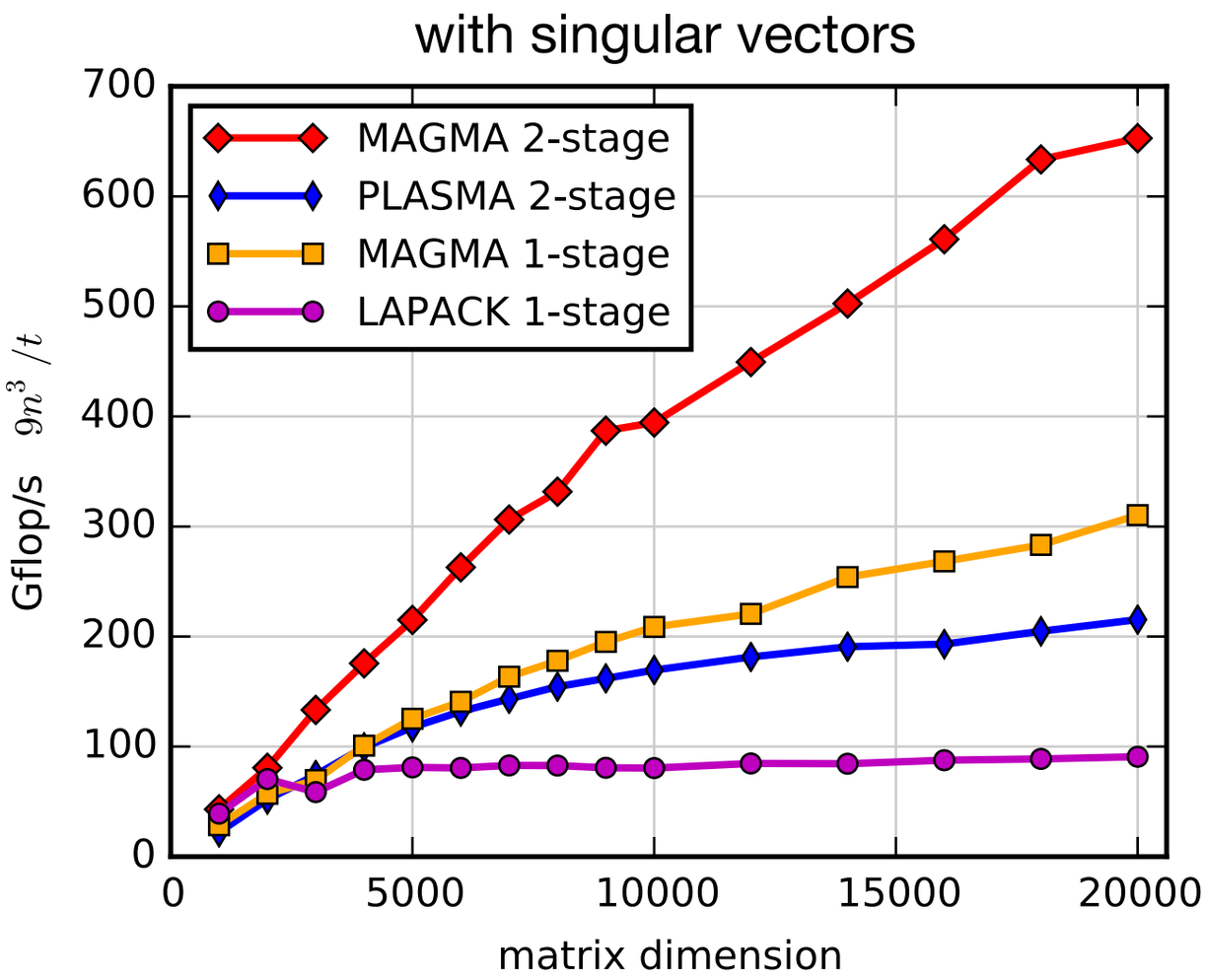
ICL

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Overall performance

higher is better





NVIDIA Pascal P100
Intel Haswell 2x10 core 2.3 GHz

Questions?