Communication avoiding algorithms in linear algebra

Laura Grigori

Alpines

INRIA Paris - LJLL, UPMC

https://who.rocq.inria.fr/Laura.Grigori/teaching.html

December 2021

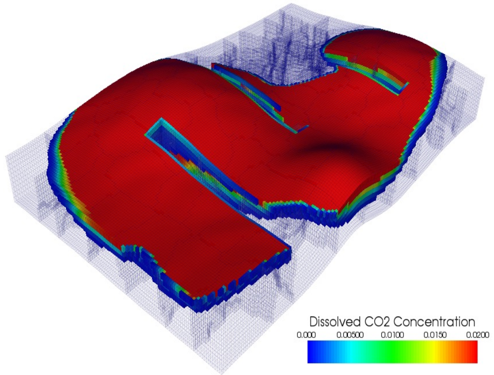
Plan

- Motivation
- Selected past work on reducing communication
- Communication complexity of linear algebra operations
- Communication avoiding for dense linear algebra
    - LU, QR, Rank Revealing QR factorizations
    - Progressively implemented in ScaLAPACK, LAPACK
    - Algorithms for multicore processors
- Conclusions

Data driven science

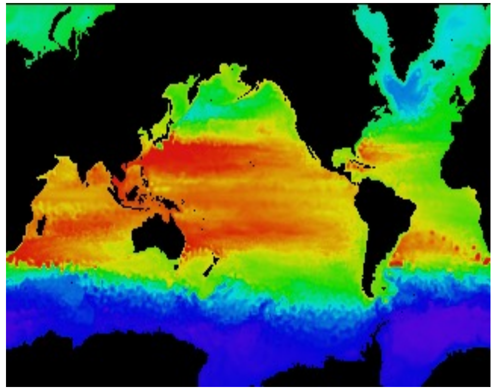
Numerical simulations require increasingly computing power as data sets grow exponentially

CO2 Underground storage



Source: T. Guignon, IFPEN

Climate modeling



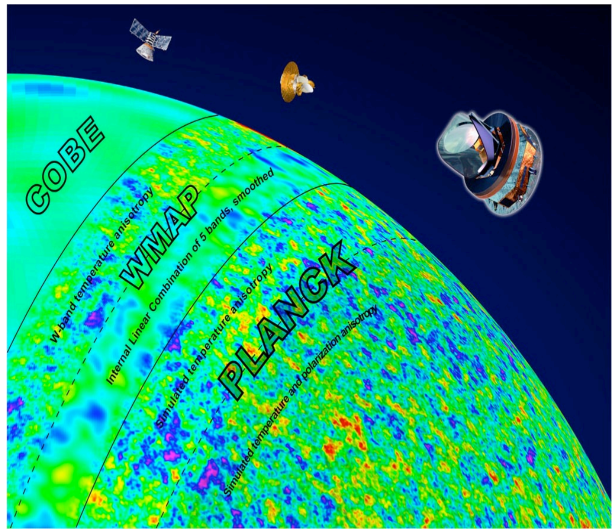
http://www.epm.ornl.gov/chammp/chammp.html

Figures from astrophysics:

- Produce and analyze multi-frequency 2D images of the universe when it was 5% of its current age.
- COBE (1989) collected 10 gigabytes of data, required 1 Teraflop per image analysis.
- PLANCK (2010) produced 1 terabyte of data, requires 100 Petaflops per image analysis.
- Future experiment (2020) estimated to collect .5 petabytes, require 100 Exaflops per image analysis.

Source: J. Borrill, LBNL, R. Stompor, Paris 7

Astrophysics: CMB data analysis



http://www.scidacreview.org/0704/html/cmb.html

CMB data analysis in an (algebraic) nutshell

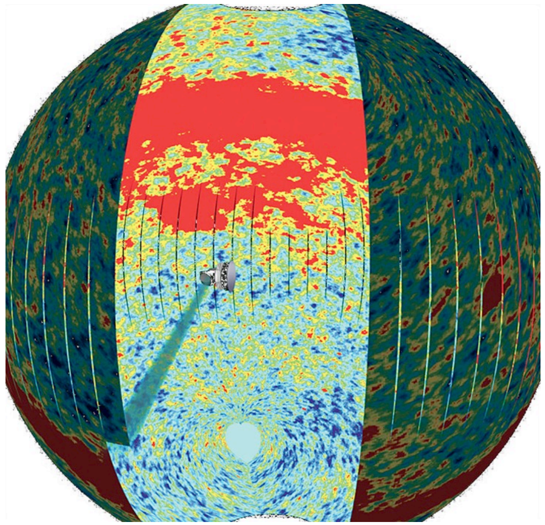
- CMB DA is a juxtaposition of the same algebraic operations
- Map-making problem
  - Find the best map x from observations d, scanning strategy A, and noise n_t

$$d = Ax + n_t$$

  - Assuming the noise properties are Gaussian and piece-wise stationary, the covariance matrix is $N = \langle n_t n_t^T \rangle$, and N^{-1} is a block diagonal symmetric Toeplitz matrix.
  - The solution of the generalized least squares problem is found by solving

$$A^T N^{-1} A x = A^T N^{-1} d$$

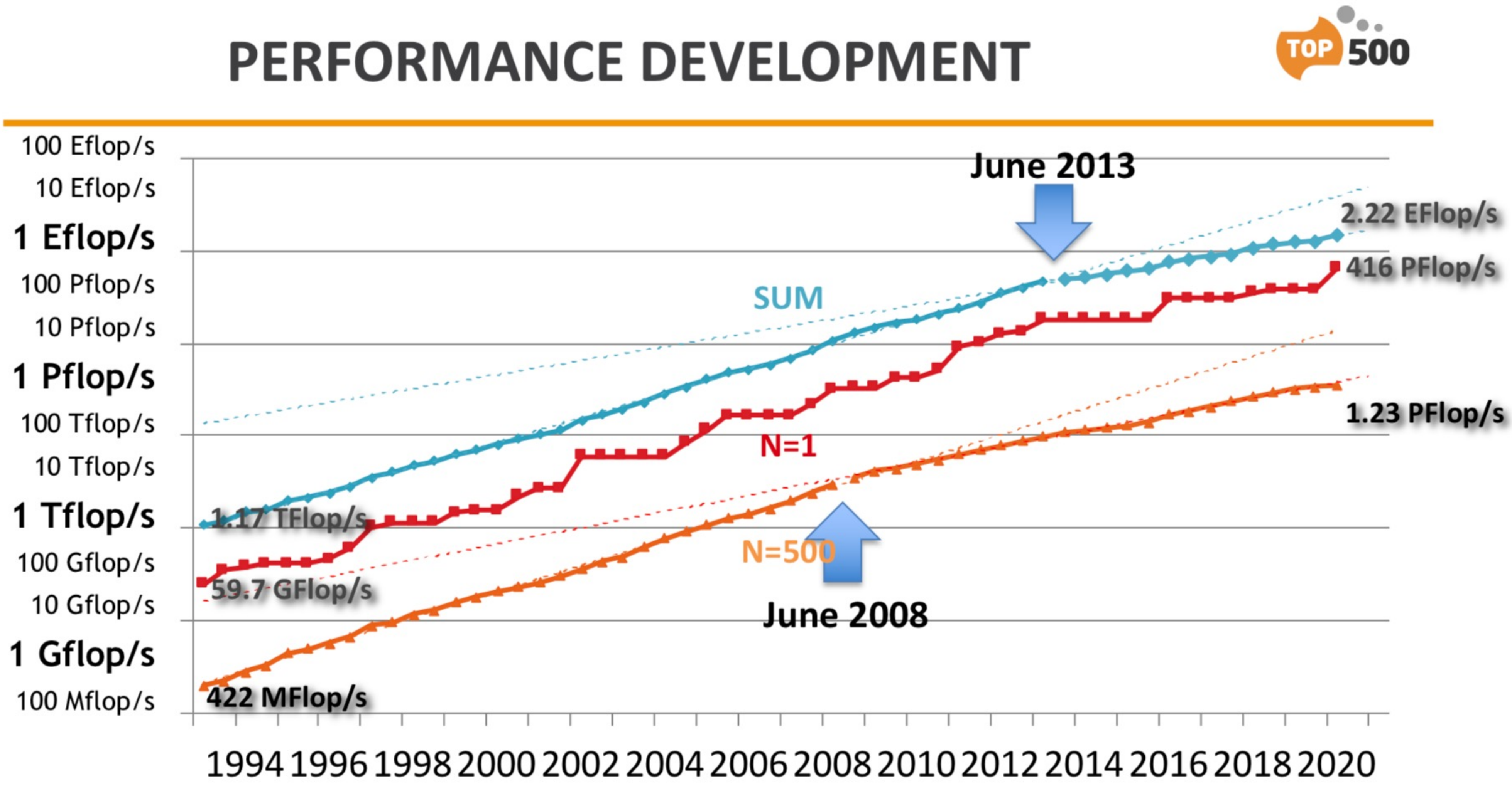
- Spherical harmonic transform (SHT)
  - Synthesize a sky image from its harmonic representation



- What is difficult about the CMB DA then ?

  Well, the data is BIG !

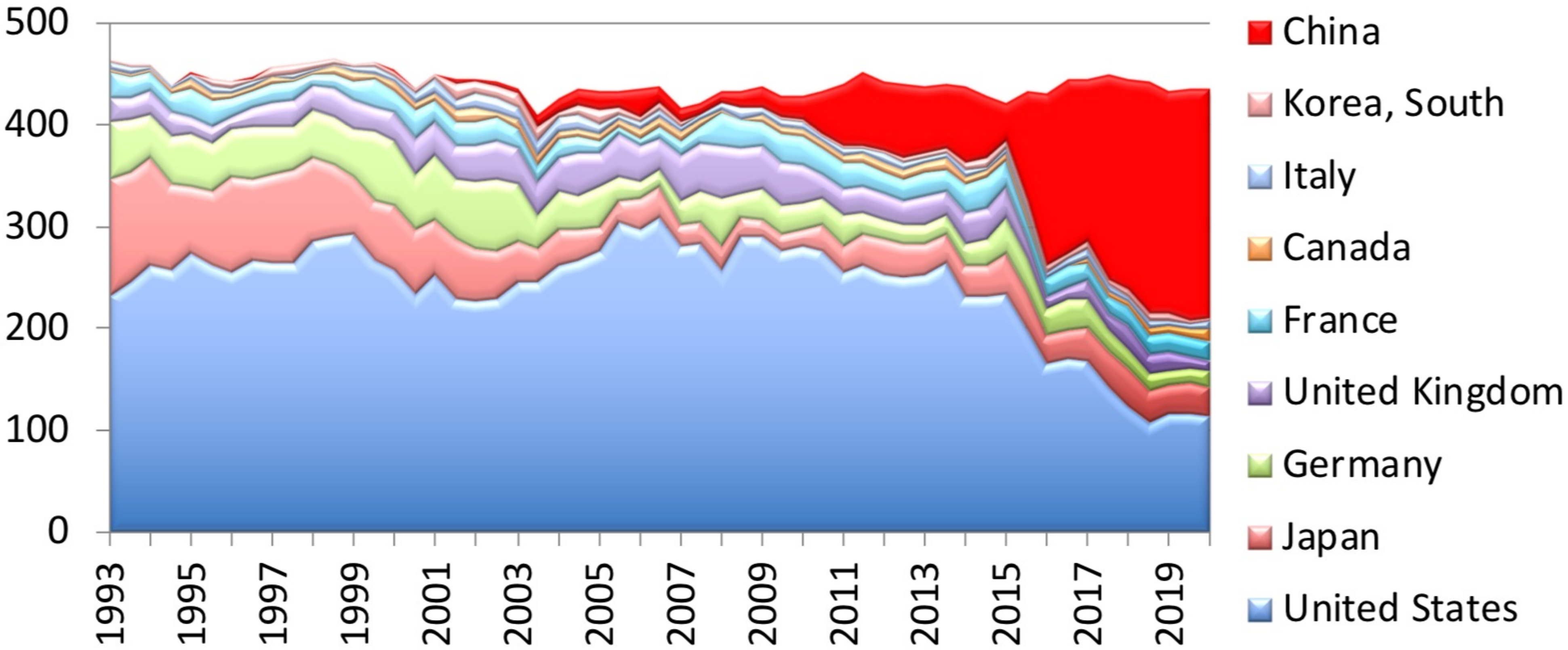
The TOP5 of the Top500, June 2020 performance development



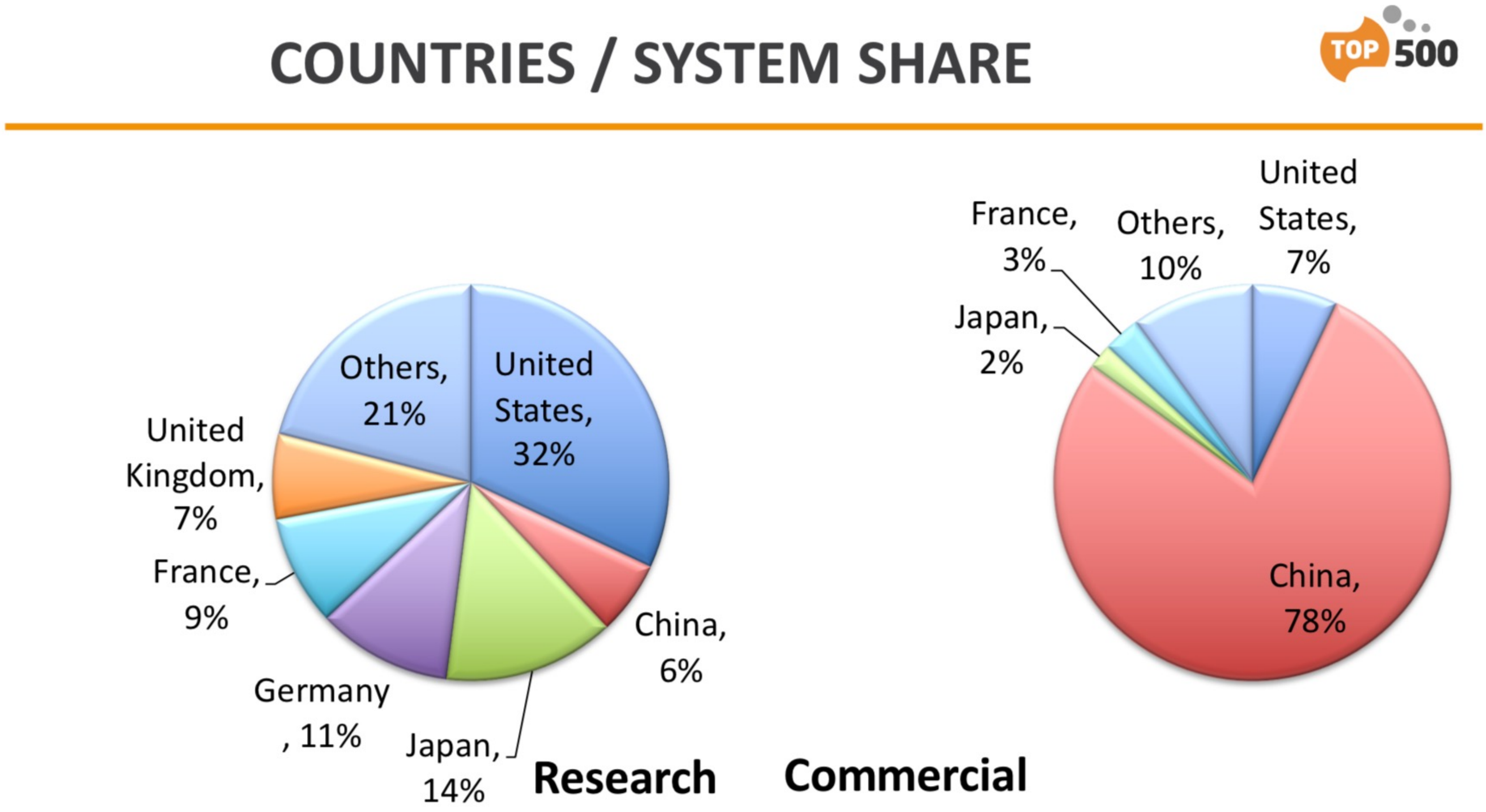
TOP10 of the Top500, June 2020

#	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	RIKEN Center for Computational Science	Fujitsu	Fugaku Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	Japan	7,299,072	415.5	28.3
2	Oak Ridge National Laboratory	IBM	Summit IBM Power System, P9 22C 3.07GHz, Mellanox EDR, NVIDIA GV100	USA	2,414,592	148.6	10.1
3	Lawrence Livermore National Laboratory	IBM	Sierra IBM Power System, P9 22C 3.1GHz, Mellanox EDR, NVIDIA GV100	USA	1,572,480	94.6	7.4
4	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
5	National University of Defense Technology	NUDT	Tianhe-2A ANUDT TH-IVB-FEP, Xeon 12C 2.2GHz, Matrix-2000	China	4,981,760	61.4	18.5
6	Eni S.p.A	Dell EMC	HPC5 PowerEdge C4140, Xeon 24C 2.1GHz, NVIDIA T. V100, Mellanox HDR	Italy	669,760	35.5	2.25
7	NVIDIA Corporation	NVIDIA	Selene DGX A100 SuperPOD, AMD 64C 2.25GHz, NVIDIA A100, Mellanox HDR	USA	277,760	27.6	1.34
8	Texas Advanced Computing Center / Univ. of Texas	Dell	Frontera Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox HDR	USA	448,448	23.5	
9	CINECA	IBM	Marconi-100 IBM Power System AC922, P9 16C 3GHz, Nvidia Volta V100, Mellanox EDR	Italy	347,776	21.6	1.98
10	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, NVIDIA Tesla P100, Aries	Switzerland	387,872	21.2	2.38

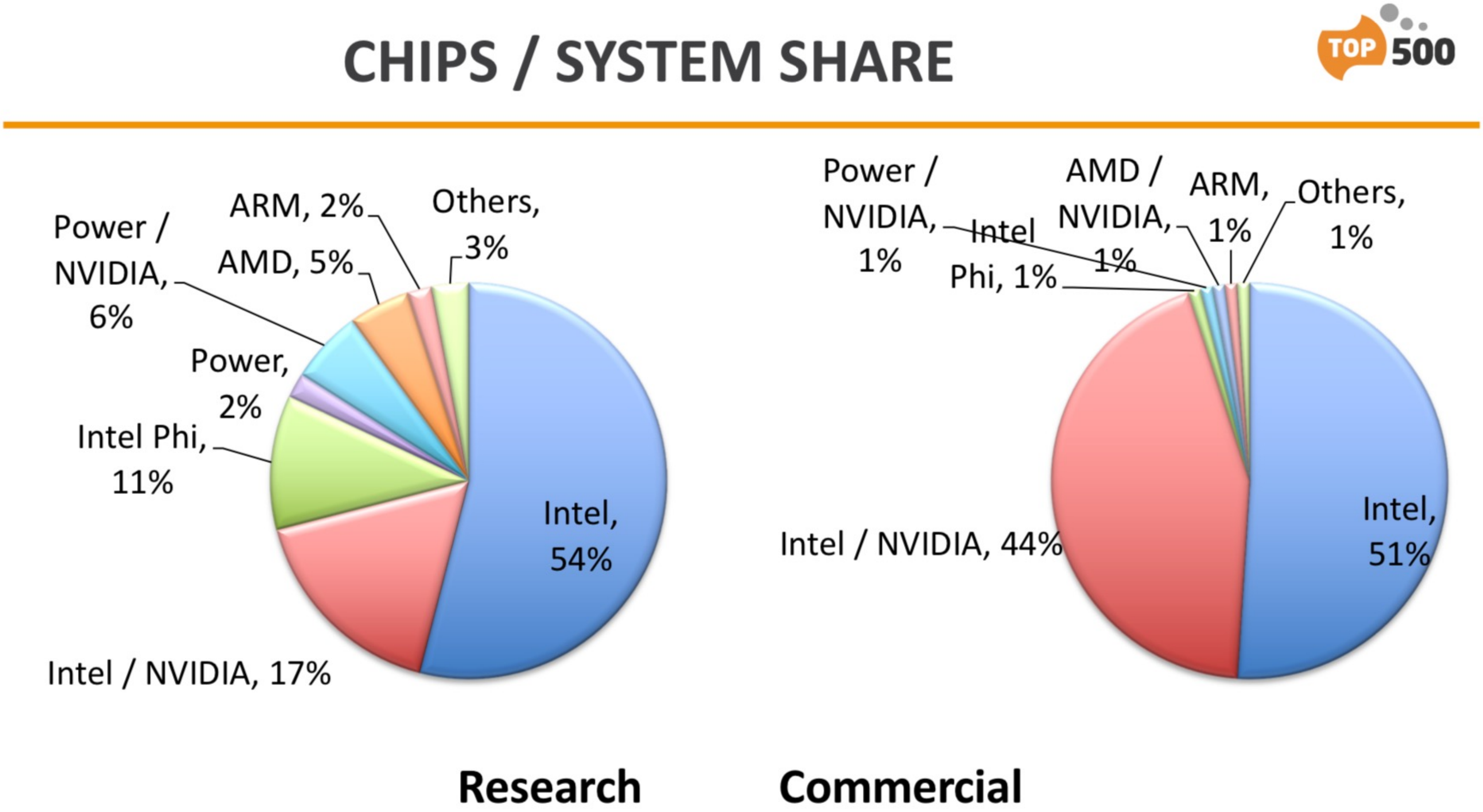
Countries, June 2020



Countries, June 2020



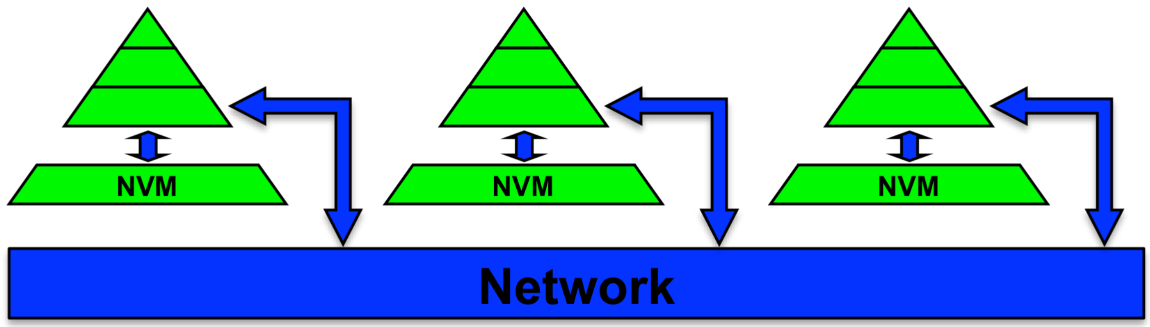
Countries, June 2020



Motivation - the communication wall

- Runtime of an algorithm is the sum of:
  - #flops x **time_per_flop**
  - #words_moved / **bandwidth**
  - #messages x **latency**

- Time to move data >> time per flop
  - Gap steadily and exponentially growing over time



Motivation - the communication wall

- Runtime of an algorithm is the sum of:
  - #flops x **time_per_flop**
  - #words_moved / **bandwidth**
  - #messages x **latency**

- Time to move data >> time per flop
  - Gap steadily and exponentially growing over time

Annual improvements			
Time/flop		Bandwidth	Latency
59%	Network	**26%**	**15%**
	DRAM	**23%**	**5%**

- Performance of an application is less than 10% of the peak performance

*"We are going to hit the **memory wall**, unless something basic changes"*
[W. Wulf, S. McKee, 95]

Compelling numbers (1)

DRAM bandwidth:

- Mid 90's ~ 0.2 bytes/flop – 1 byte/flop
- Past few years ~ 0.02 to 0.05 bytes/flop

DRAM latency:

- DDR2 (2007) ~ 120 ns 1x
- DDR4 (2014) ~ 45 ns 2.6x in 7 years
- Stacked memory ~ similar to DDR4 13% / year

Time/flop

- 2006 Intel Yonah ~ 2GHz x 2 cores (32 GFlops/chip) 1x
- 2015 Intel Haswell ~2.3GHz x 16 cores (588 GFlops/chip) 18x in 9 years

34% / year

Source: J. Shalf, LBNL

Compelling numbers (2)

Fetch from DRAM 1 byte of data

- 1988: compute 6 flops
- 2004: compute a few 100 flops
- 2015: compute 26460 flops/chip (see below)

Receive from another proc 1 byte of data:

- Compute 147000 - 1065000 flops

Example of one supercomputer:

- Intel Haswell: 8 flops per cycle per core
- Interconnect: 0.25 μs to 3.7 μs MPI latency, 8GB/sec MPI bandwidth

The role of numerical linear algebra

- Challenging applications often rely on solving linear algebra problems
- Linear systems of equations

  Solve $Ax = b$, where $A \in \mathbf{R}^{nxn}$, $b \in \mathbf{R}^{n}$, $x \in \mathbf{R}^{n}$

  - Direct methods

    $PA = LU$, then solve $P^{T}LUx = b$

    LU factorization is backward stable,

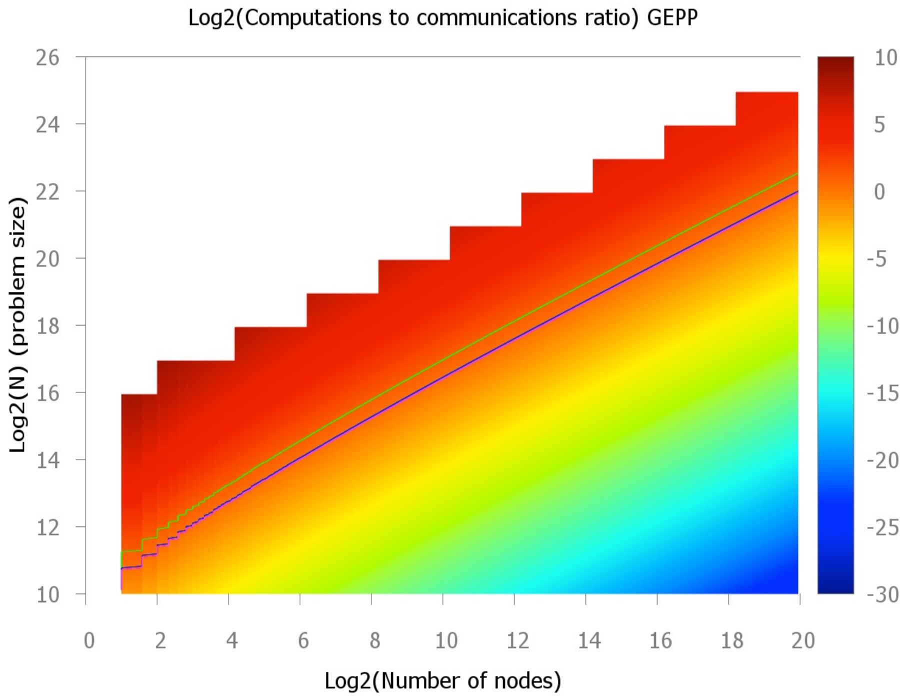
  - Iterative methods
    - Find a solution x_k from $x_0 + K_k (A, r_0)$, where $K_k (A, r_0) = span \{r_0, A r_0, \ldots, A^{k-1} r_0\}$ such that the Petrov-Galerkin condition $b - Ax_k \perp L_k$ is satisfied,

      where L_k is a subspace of dimension k and $r_0 = Ax_0 - b$.
    - Convergence depends on $\kappa(A)$ and the eigenvalue distribution (for SPD matrices).

Approaches for reducing communication

- Tuning
  - Overlap communication and computation, at most a factor of 2 speedup

- Same numerical algorithm,
  different schedule of the computation
  - Block algorithms for NLA
    - Barron and Swinnerton-Dyer, 1960
    - ScaLAPACK, Blackford et al 97
  - Cache oblivious algorithms for NLA
    - Gustavson 97, Toledo 97, Frens and Wise 03, Ahmed and Pingali 00



- Same algebraic framework, different numerical algorithm
  - The approach used in CA algorithms
  - More opportunities for reducing communication, may affect stability

Selected past work on reducing communication

- Only few examples shown, many references available

A. Tuning

- Overlap communication and computation, at most a factor of 2 speedup

B. Ghosting

- Standard approach in *explicit methods*
- Store redundantly data from neighboring processors for future computations

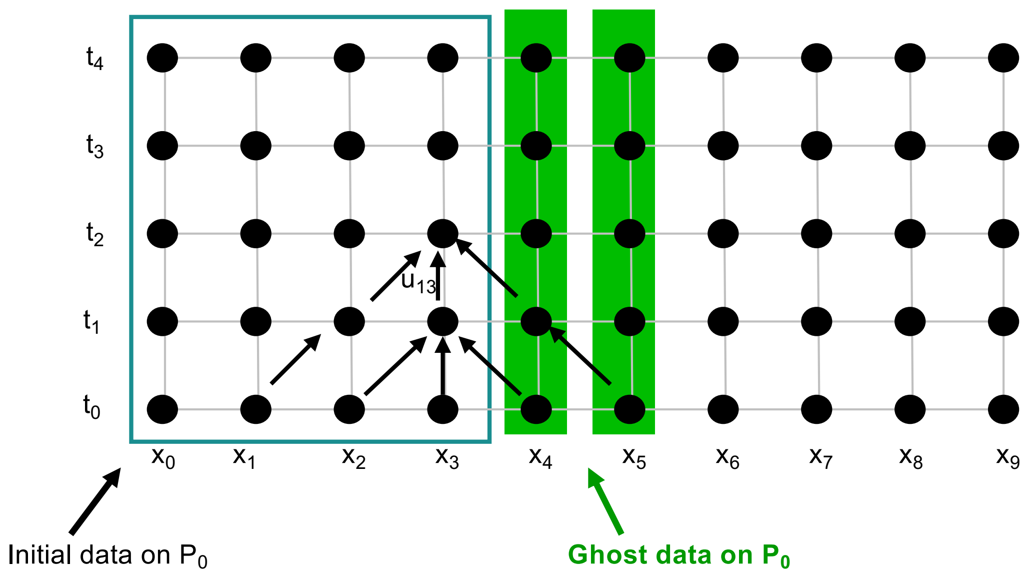
Example of a parabolic PDE

$$u_t = \alpha \, \Delta u$$

with a finite difference,

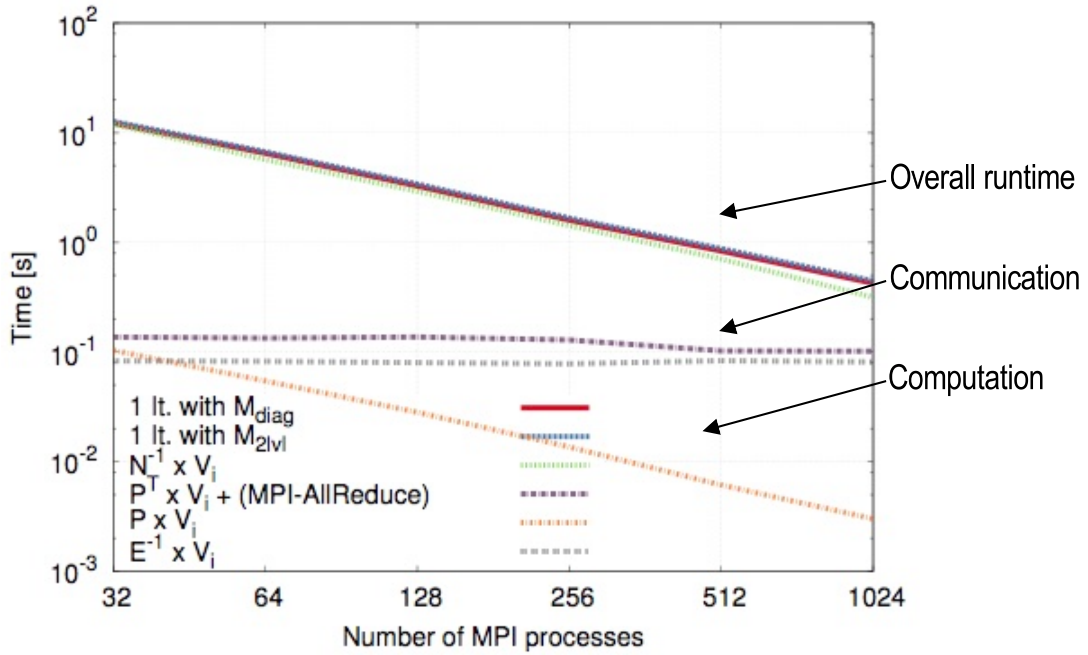
the solution at a grid point is:

$$u_{i,j+1} = u(x_i, t_{j+1}) = f(u_{i-1,j}, u_{ij}, u_{i+1,j})$$

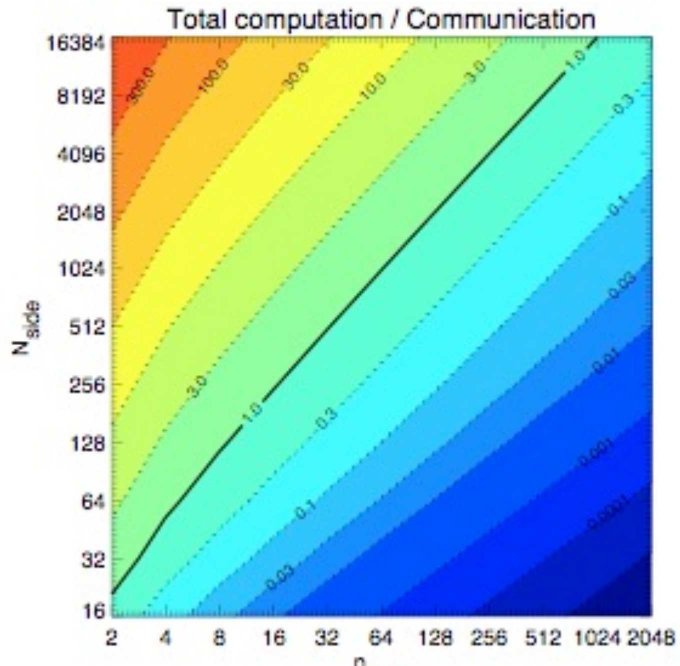


Communication in CMB data analysis

- Map-making problem
  - Find the best map x from observations d, scanning strategy A, and noise N^{-1}
  - Solve generalized least squares problem involving sparse matrices of size 10^{12}-by-10^{7}
- Spherical harmonic transform (SHT)
  - Synthesize a sky image from its harmonic representation
    - Computation over rows of a 2D object (summation of spherical harmonics)
    - Communication to transpose the 2D object
    - Computation over columns of the 2D object (FFTs)



Map making, with R. Stompor, M. Szydlarski
Results obtained on Hopper, Cray XE6, NERSC

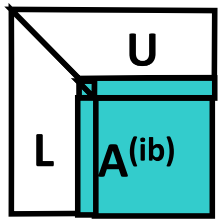


SHT, with R. Stompor, M. Szydlarski
Simulation on a petascale computer

Evolution of numerical libraries

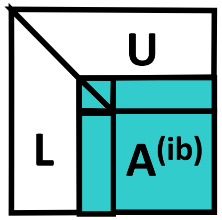
LINPACK (70's)

- vector operations, uses BLAS1/2
- HPL benchmark based on Linpack LU factorization



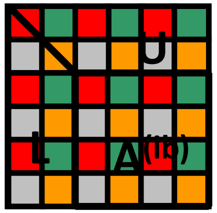
LAPACK (80's)

- Block versions of the algorithms used in LINPACK
- Uses BLAS3



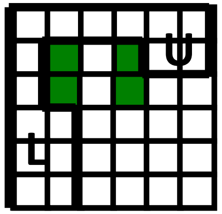
ScaLAPACK (90's)

- Targets distributed memories
- 2D block cyclic distribution of data
- PBLAS based on message passing



PLASMA (2008): new algorithms

- Targets many-core
- Block data layout
- Low granularity, high asynchronicity

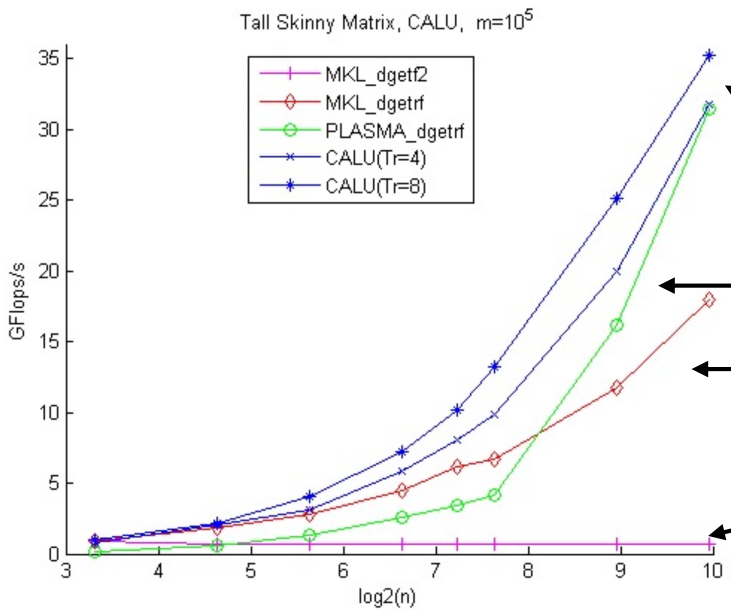


Project developed by U Tennessee Knoxville, UC Berkeley, other collaborators.

Source: inspired from J. Dongarra, UTK, J. Langou, CU Denver

Evolution of numerical libraries

- Did we need new algorithms?
  - Results on two-socket, quad-core Intel Xeon EMT64 machine, 2.4 GHz per core, peak performance 76.5 Gflops/s
  - LU factorization of an m-by-n matrix, $m=10^5$ and n varies from 10 to 1000



LU using communication avoiding approach

LU from PLASMA using operations on tiles

LU from Intel MKL using lapack block operations
Lapack will not get even 1/2 of this performance

LU using vector operations

Motivation

- The communication problem needs to be taken into account higher in the computing stack
- A paradigm shift in the way the numerical algorithms are devised is required
- Communication avoiding algorithms - a novel perspective for numerical linear algebra
  - Minimize volume of communication
  - Minimize number of messages
  - Minimize over multiple levels of memory/parallelism
  - Allow redundant computations (preferably as a low order term)

Communication Complexity of Dense Linear Algebra

- Matrix multiply, using $2n^3$ flops (sequential or parallel)
  - Hong-Kung (1981), Irony/Tishkin/Toledo (2004)
  - Lower bound on Bandwidth = Ω (#flops / $M^{1/2}$)
  - Lower bound on Latency = Ω (#flops / $M^{3/2}$)

- Same lower bounds apply to LU using reduction
  - Demmel, LG, Hoemmen, Langou 2008

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

- And to almost all direct linear algebra [Ballard, Demmel, Holtz, Schwartz, 09]

Lower bounds for linear algebra

- Computation modelled as an n-by-n-by-n set of lattice points
  (i,j,k) represents the operation $c(i,j)\ {+}{=}\ f_{ij}(\ g_{ijk}\ (\ a(i,k)*b(k,j))\ )\ )$
- The computation is divided in S phases
- Each phase contains exactly M (the fast memory size) load and store instructions
- Determine how many flops the algorithm can compute in each phase, by applying discrete Loomis-Whitney inequality:

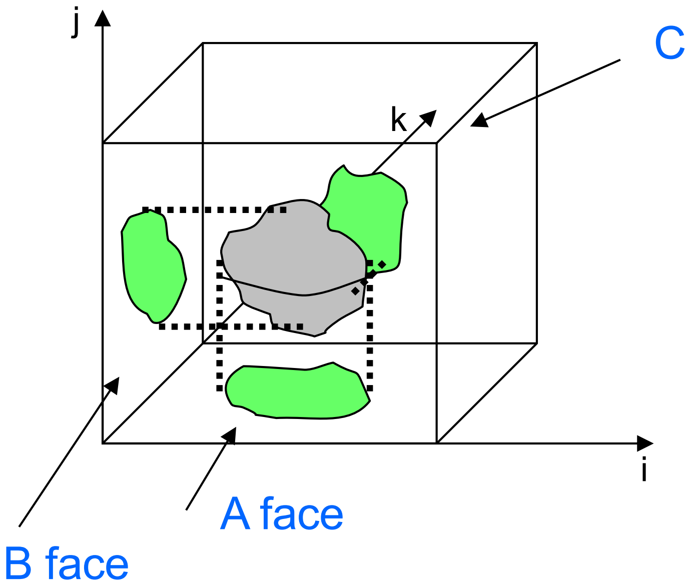
$$w^2 \le N_A N_B N_C$$

Algorithms in direct linear alg

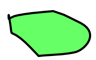
$for\ i,j,k = 1:n$

$\quad c(i,j) = f_{ij}(g_{ijk}(a(i,k),b(k,j)))$

$endfor$



 - set of points in R^3, represent w arithmetics

 - orthogonal projections of the points onto coordinate planes N_A, N_B, N_C represent values of A, B, C

Lower bounds for matrix multiplication (contd)

-
- Discrete Loomis-Whitney inequality:

$$w^2 \leq N_A N_B N_C$$

- Since there are at most 2M elements of A, B, C in a phase, the bound is:

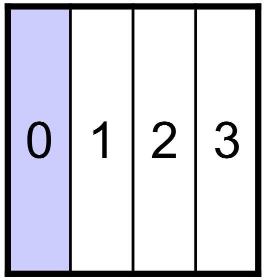
$$w \leq 2\sqrt{2}M^{3/2}$$

- The number of phases S is #flops/w, and hence the lower bound on communication is:

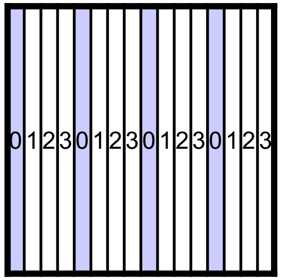
$$\#messages \geq \frac{\#flops}{w} = \Omega\left(\frac{\#flops}{M^{\frac{3}{2}}}\right)$$

$$\#loads/stores \geq \Omega\left(\frac{\#flops}{M^{1/2}}\right)$$

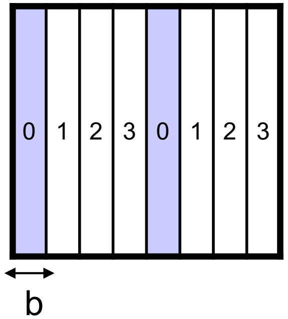
Matrix distributions



1) 1D Column Blocked Layout

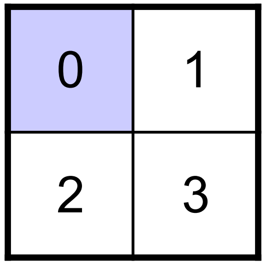


2) 1D Column Cyclic Layout

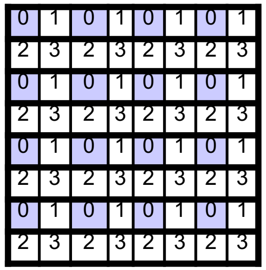


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout



Generalizes others

6) 2D Row and Column Block Cyclic Layout

Source slide: J. Demmel

MatMul with 2D Layout

- Consider processors in 2D grid (physical or logical)
- Processors can communicate with 4 nearest neighbors
  - Broadcast along rows and columns

$$\begin{array}{|c|c|c|}\hline p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \hline \end{array}$$

- Assume p processors form square s x s grid, $s = p^{1/2}$

Source slide: J. Demmel

Cannon's Algorithm

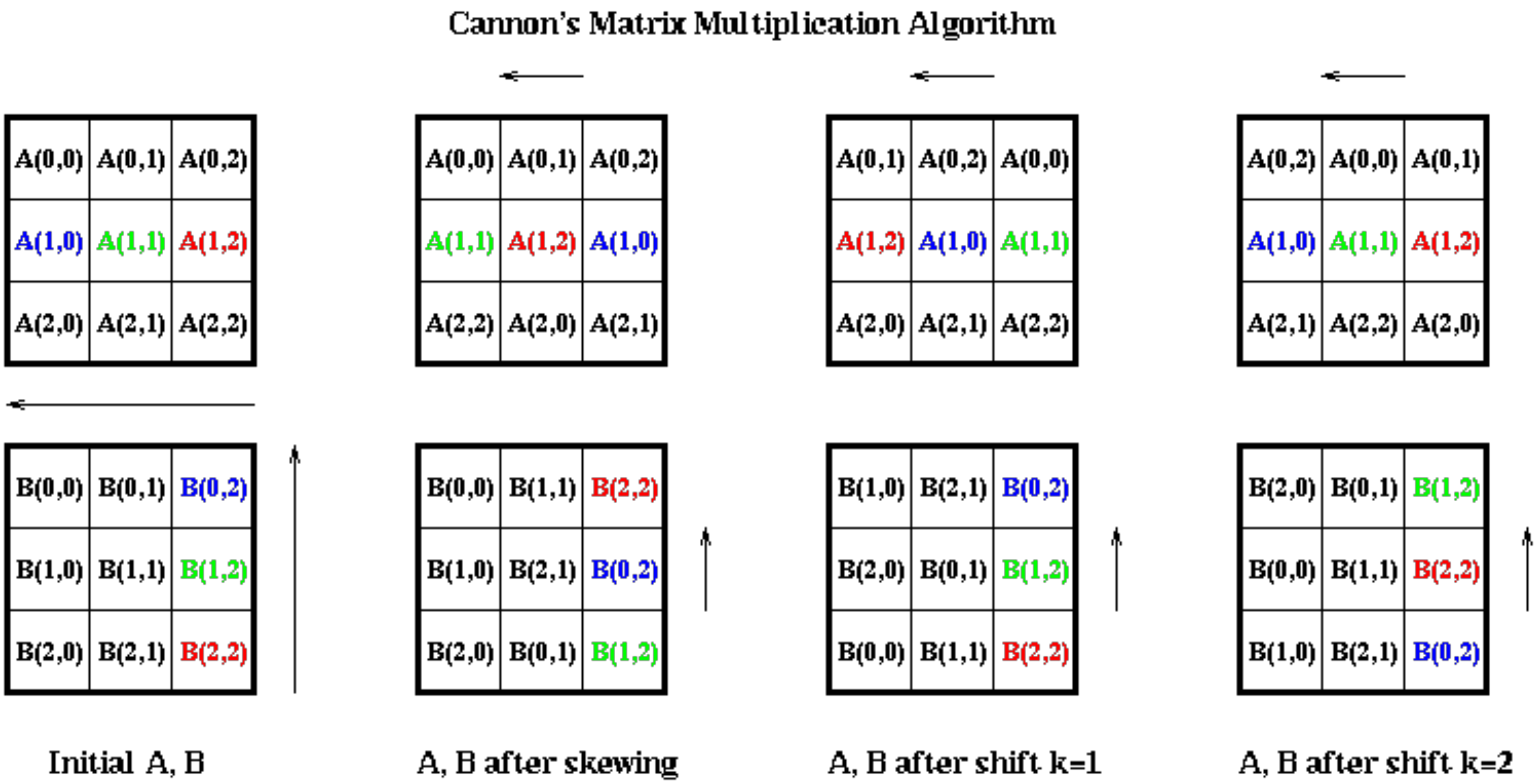
… $C(i,j) = C(i,j) + \sum_k A(i,k)*B(k,j)$

… assume s = sqrt(p) is an integer

```
forall  i=0 to s-1                … "skew" A
     left-circular-shift row i of A by i
     … so that A(i,j) overwritten by A(i,(j+i)mod s)
forall  i=0 to s-1                … "skew" B
     up-circular-shift column i of B by i
     … so that B(i,j) overwritten by B((i+j)mod s), j)
for k=0 to s-1          … sequential
     forall i=0 to s-1 and j=0 to s-1    … all processors in parallel
          C(i,j) = C(i,j) + A(i,j)*B(i,j)
          left-circular-shift each row of A by 1
          up-circular-shift each column of B by 1
```

Source slide: J. Demmel

Cannon's Matrix Multiplication



C(1,2) = A(1,0) * B(0,2) + A(1,1) * B(1,2) + A(1,2) * B(2,2)

Source slide: J. Demmel

Cost of Cannon's Algorithm

forall i=0 to s-1 … recall s = sqrt(p)

left-circular-shift row i of A by i … cost $\le$ s*(α + β*n^2/p)

forall i=0 to s-1

up-circular-shift column i of B by i … cost $\le$ s*(α + β*n^2/p)

for k=0 to s-1

forall i=0 to s-1 and j=0 to s-1

C(i,j) = C(i,j) + A(i,j)*B(i,j) … cost = 2*$(n/s)^3$ = 2*$n^3/p^{3/2}$

left-circular-shift each row of A by 1 … cost = α + β*n^2/p

up-circular-shift each column of B by 1 … cost = α + β*n^2/p

- Total Time = 2*n^3/p + 4* s*α + 4*β*n^2/s - Optimal!
- Parallel Efficiency = 2*n^3 / (p * Total Time)

  = 1/(1 + α * 2*$(s/n)^3$ + β * 2*(s/n))

  = 1/(1 + O(sqrt(p)/n))
- Grows to 1 as n/s = n/sqrt(p) = sqrt(data per processor) grows

Source slide: J. Demmel

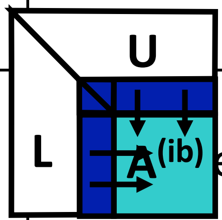
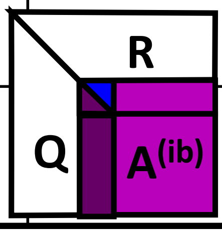
Sequential algorithms and communication bounds

Algorithm	Minimizing #words (not #messages)	Minimizing #words and #messages
Cholesky	LAPACK	[Gustavson, 97] [Ahmed, Pingali, 00]
LU	LAPACK (few cases) [Toledo,97], [Gustavson, 97] both use partial pivoting	[LG, Demmel, Xiang, 08] [Khabou, Demmel, LG, Gu, 12] uses tournament pivoting
QR	LAPACK (few cases) [Elmroth,Gustavson,98]	[Frens, Wise, 03], 3x flops [Demmel, LG, Hoemmen, Langou, 08] [Ballard et al, 14]
RRQR		[Demmel, LG, Gu, Xiang 11] uses tournament pivoting, 3x flops

- Only several references shown for block algorithms (LAPACK), cache-oblivious algorithms and communication avoiding algorithms
- CA algorithms exist also for SVD and eigenvalue computation

2D Parallel algorithms and communication bounds

- If memory per processor = n^2 / P, the lower bounds become
  #words_moved $\geq \Omega(n^2 / P^{1/2})$, #messages $\geq \Omega(P^{1/2})$

Algorithm	Minimizing #words (not #messages)	Minimizing #words and #messages
Cholesky	ScaLAPACK	ScaLAPACK
LU 	ScaLAPACK es partial pivoting	[LG, Demmel, Xiang, 08] [Khabou, Demmel, LG, Gu, 12] uses tournament pivoting
QR 	ScaLAPACK	[Demmel, LG, Hoemmen, Langou, 08] [Ballard et al, 14]
RRQR	ScaLAPACK	[Demmel, LG, Gu, Xiang 13] uses tournament pivoting, 3x flops

- Only several references shown, block algorithms (ScaLAPACK) and communication avoiding algorithms
- CA algorithms exist also for SVD and eigenvalue computation

The algebra of LU factorization

- Compute the factorization PA = LU

- Given the matrix

$$A = \begin{pmatrix} 3 & 1 & 3 \\ 6 & 7 & 3 \\ 9 & 12 & 3 \end{pmatrix}$$

Let

$$M_1 = \begin{pmatrix} 1 & & \\ -2 & 1 & \\ -3 & & 1 \end{pmatrix}, \qquad M_1 A = \begin{pmatrix} 3 & 1 & 3 \\ 0 & 5 & -3 \\ 0 & 9 & -6 \end{pmatrix}$$

The algebra of LU factorization (contd)

- In general

$$
\begin{aligned}
A^{(k+1)} &= M_k A^{(k)} := \begin{pmatrix} I_{k-1} & & & & \\ & 1 & & & \\ & -m_{k+1,k} & 1 & & \\ & \cdots & & \ddots & \\ & -m_{n,k} & & & 1 \end{pmatrix} A^{(k)}, \text{ where} \\
M_k &= I - m_k e_k^T, \quad M_k^{-1} = I + m_k e_k^T
\end{aligned}
$$

where e_k is the k-th unit vector, $e_i^T m_k = 0, \forall i \leq k$

- The factorization can be written as

$$M_{n-1} \ldots M_1 A = A^{(n)} = U$$

The algebra of LU factorization (contd)

We obtain

$$
\begin{aligned}
A &= M_1^{-1} \dots M_{n-1}^{-1} U \\
&= (I + m_1 e_1^T) \dots (I + m_{n-1} e_{n-1}^T) U \\
&= \left(I + \sum_{i=1}^{n-1} m_i e_i^T \right) U \\
&= \begin{pmatrix} 1 & & & \\ m_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ m_{n1} & m_{n2} & \dots & 1 \end{pmatrix} U = LU
\end{aligned}
$$

The need for pivoting

- For stability avoid division by small elements, otherwise ||A-LU|| can be large
  - Because of roundoff error
- For example

$$A = \begin{pmatrix} 0 & 3 & 3 \\ 3 & 1 & 3 \\ 6 & 2 & 3 \end{pmatrix}$$

has an LU factorization if we permute the rows of A

$$PA = \begin{pmatrix} 6 & 2 & 3 \\ 0 & 3 & 3 \\ 3 & 1 & 3 \end{pmatrix} = \begin{pmatrix} 1 & & \\ & 1 & \\ 0.5 & & 1 \end{pmatrix} \begin{pmatrix} 6 & 2 & 3 \\ & 3 & 3 \\ & & 1.5 \end{pmatrix}$$

- Partial pivoting allows to bound all elements of L by 1.

LU with partial pivoting – BLAS 2 algorithm

- Algorithm for computing the in place LU factorization of a matrix of size $n \times n$.
- $\#flops = 2n^3/3$

for k = 1:n-1 **do**
    Let a_{ik} be the element of maximum magnitude in $A(k:n,k)$
    Permute row i and row k
    $A(k+1:n,k) = A(k+1:n,k)/a_{kk}$
    for $i = k+1:n$ **do**
        for $j = k+1:n$ **do**
            $a_{ij} = a_{ij} - a_{ik}a_{kj}$
        end for
    end for
end for

Block LU factorization – obtained by delaying updates

- Matrix A of size *nxn* is partitioned as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ where } A_{11} \text{ is } b \times b$$

- The first step computes LU with partial pivoting of the first block:

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11}$$

- The factorization obtained is:

$$P_1 A = \begin{pmatrix} L_{11} & \\ L_{21} & I_{n-b} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & A_{22}^1 \end{pmatrix}, \text{ where } \begin{matrix} U_{12} = L_{11}^{-1} A_{12} \\ A_{22}^1 = A_{22} - L_{21} U_{12} \end{matrix}$$

- The algorithm continues recursively on the trailing matrix A_{22}^1

Block LU factorization – the algorithm

1. Compute LU with partial pivoting of the first panel

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11}$$

2. Pivot by applying the permutation matrix P_1 on the entire matrix

$$P_1 A = \overline{A}$$

3. Solve the triangular system to compute a block row of U

$$U_{12} = L_{12}^{-1} \overline{A}_{12}$$

4. Update the trailing matrix

$$\overline{A}_{22}^{1} = \overline{A}_{22} - L_{21} U_{12}$$

1. The algorithm continues recursively on the trailing matrix $\overline{A}_{22}^{1}$

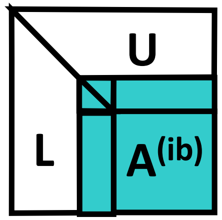
LU factorization (as in ScaLAPACK pdgetrf)

LU factorization on a $P = P_r \times P_c$ grid of processors

For ib = 1 to n-1 step b

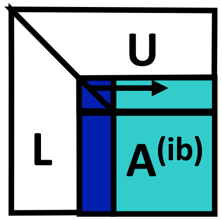
$A^{(ib)} = A(ib{:}n, ib{:}n)$

#messages



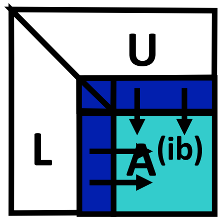
(1) Compute panel factorization $O(n \log_2 P_r)$

- find pivot in each column, swap rows



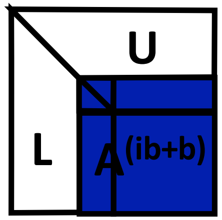
(2) Apply all row permutations $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast pivot information along the rows
- swap rows at left and right



(3) Compute block row of U $O(n/b \log_2 P_c)$

- broadcast right diagonal block of L of current panel



(4) Update trailing matrix $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast right block column of L
- broadcast down block row of U

General scheme for QR factorization by Householder transformations

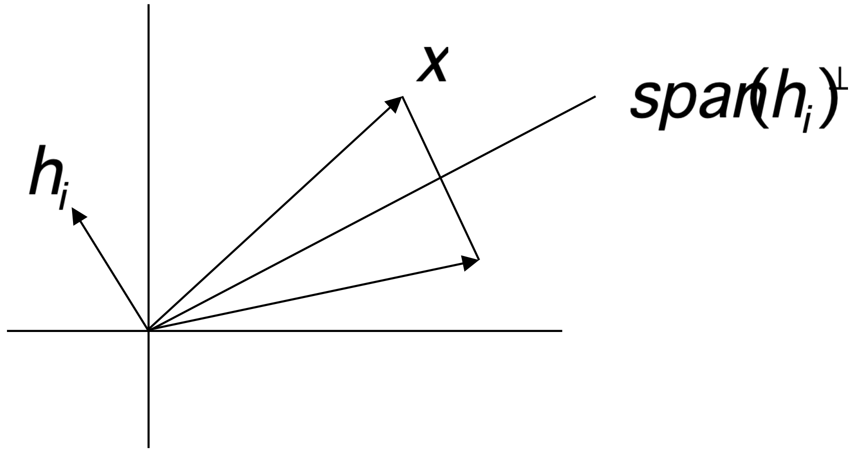
The Householder matrix

$$H_i = I - \tau_i h_i h_i^T$$

has the following properties:

- is symmetric and orthogonal,

  $H_i^2 = I$,
- is independent of the scaling of h_i,
- it reflects x about the hyperplane $span(h_i)^{\perp}$



- For QR, we choose a Householder matrix that allows to annihilate the elements of a vector x, except first element.

General scheme for QR factorization by Householder transformations

- Apply Householder transformations to annihilate subdiagonal entries

$$A = \begin{pmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{pmatrix} = H_1 \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{pmatrix} = H_1 \begin{pmatrix} 1 & \\ & \tilde{H}_2 \end{pmatrix} \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{pmatrix} =$$

$$= H_1 H_2 \begin{pmatrix} 1 & & \\ & 1 & \\ & & \tilde{H}_3 \end{pmatrix} \begin{pmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{pmatrix} = H_1 H_2 H_3 R = QR$$

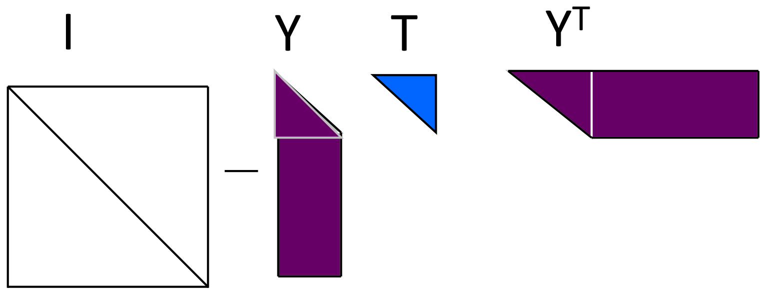
- For A of size mxn, the factorization can be written as:

$$H_n H_{n-1} \ldots H_2 H_1 A = R \rightarrow A = \left(H_n H_{n-1} \ldots H_2 H_1\right)^T R$$

$$Q = H_1 H_2 \ldots H_n$$

Compact representation for Q

- Orthogonal factor Q can be represented implicitly as



- Example for b=2:

$$Y = (h_1 | h_2), \qquad T = \begin{pmatrix} \tau_1 & -\tau_1 h_1^T h_2 \tau_2 \\ & \tau_2 \end{pmatrix}$$

Algebra of block QR factorization

Matrix A of size *nxn* is partitioned as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ where } A_{11} \text{ is } b \times b$$

Block QR algebra

The first step of the block QR factorization algorithm computes:

$$Q_1^T A = \begin{bmatrix} R_{11} & R_{12} \\ & A_{22}^1 \end{bmatrix}$$

The algorithm continues recursively on the trailing matrix A_{22}^1

Block QR factorization

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ & A_{22}^1 \end{pmatrix}$$

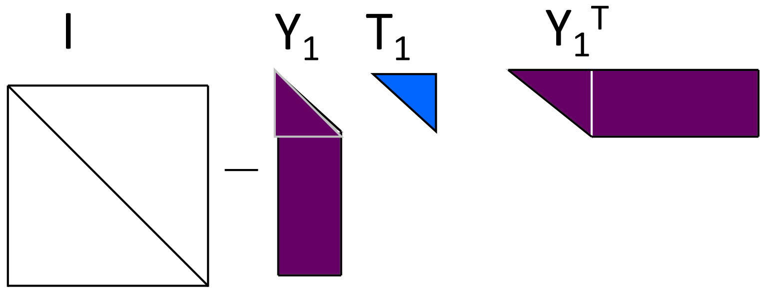
Block QR algebra:

1. Compute panel factorization:

$$\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ \\ \end{pmatrix}, \quad Q_1 = H_1 H_2 \ldots H_b$$

2. Compute the compact representation:

$$Q_1 = I - Y_1 T_1 Y_1^T$$



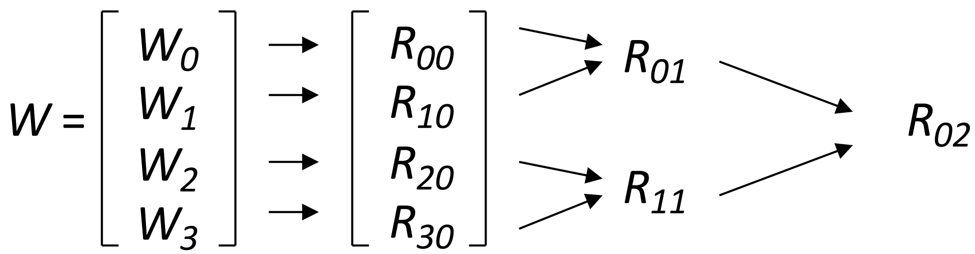
3. Update the trailing matrix:

$$\left(I - Y_1 T_1^T Y_1^T\right)\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} - Y_1\left(T_1^T\left(Y_1^T\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}\right)\right) = \begin{pmatrix} R_{12} \\ A_{22}^1 \end{pmatrix}$$

4. The algorithm continues recursively on the trailing matrix.

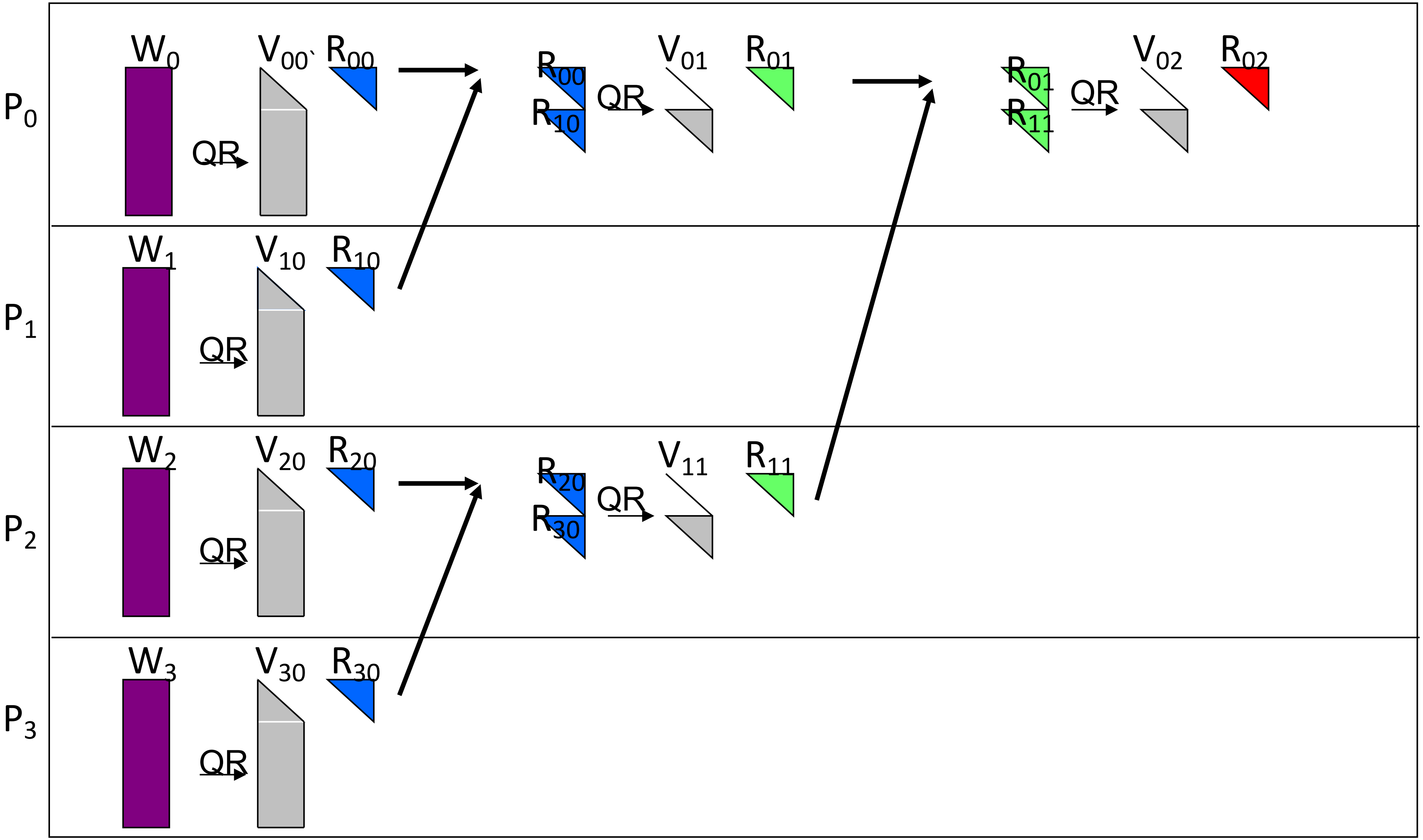
TSQR: QR factorization of a tall skinny matrix using Householder transformations

- QR decomposition of m x b matrix W, m >> b
  - P processors, block row layout
- Classic Parallel Algorithm
  - Compute Householder vector for each column
  - Number of messages $\propto$ b log P
- Communication Avoiding Algorithm
  - Reduction operation, with QR as operator
  - Number of messages $\propto$ log P



J. Demmel, LG, M. Hoemmen, J. Langou, 08

Parallel TSQR



References: Golub, Plemmons, Sameh 88, Pothen, Raghavan, 89, Da Cunha, Becker, Patterson, 02

Algebra of TSQR

Parallel:
$$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \\ \rightarrow \end{array} \begin{array}{c} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{array} \begin{array}{c} \searrow \\ \nearrow \\ \searrow \\ \nearrow \end{array} \begin{array}{c} R_{01} \\ \\ R_{11} \end{array} \begin{array}{c} \searrow \\ \nearrow \end{array} R_{02}$$

$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ \hline Q_{10}R_{10} \\ \hline Q_{20}R_{20} \\ \hline Q_{30}R_{30} \end{pmatrix} = \begin{pmatrix} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{pmatrix} \cdot \begin{pmatrix} R_{00} \\ \hline R_{10} \\ \hline R_{20} \\ \hline R_{30} \end{pmatrix}$$

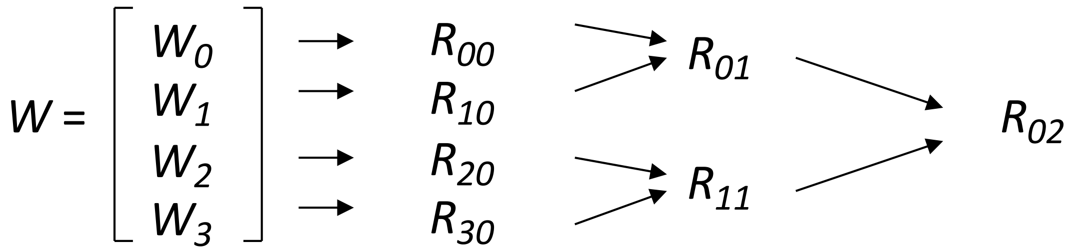
$$\begin{pmatrix} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01}R_{01} \\ \hline Q_{11}R_{11} \end{pmatrix} = \begin{pmatrix} Q_{01} & \\ \hline & Q_{11} \end{pmatrix} \cdot \begin{pmatrix} R_{01} \\ \hline R_{11} \end{pmatrix} \qquad \begin{pmatrix} R_{01} \\ \hline R_{11} \end{pmatrix} = Q_{02}R_{02}$$

Q is represented implicitly as a product

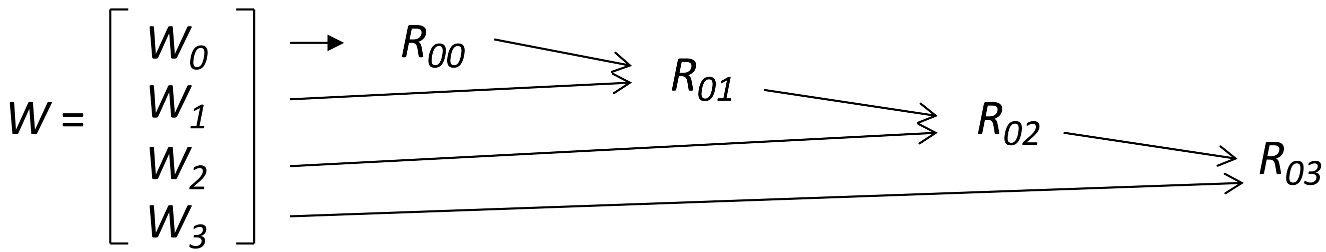
Output: $\{Q_{00}, Q_{10}, Q_{00}, Q_{20}, Q_{30}, Q_{01}, Q_{11}, Q_{02}, R_{02}\}$

Flexibility of TSQR and CAQR algorithms

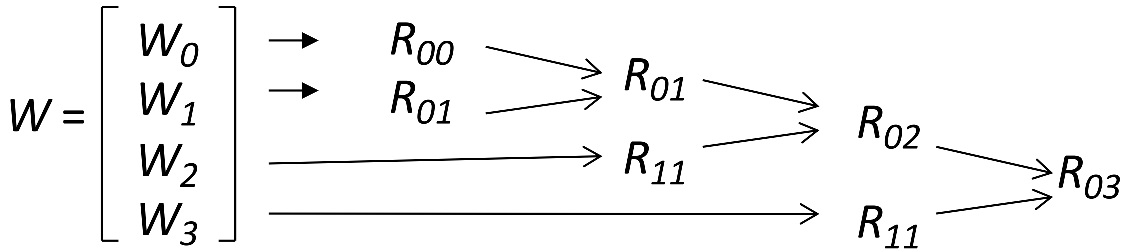
Parallel:



Sequential:



Dual Core:



Reduction tree will depend on the underlying architecture, could be chosen dynamically

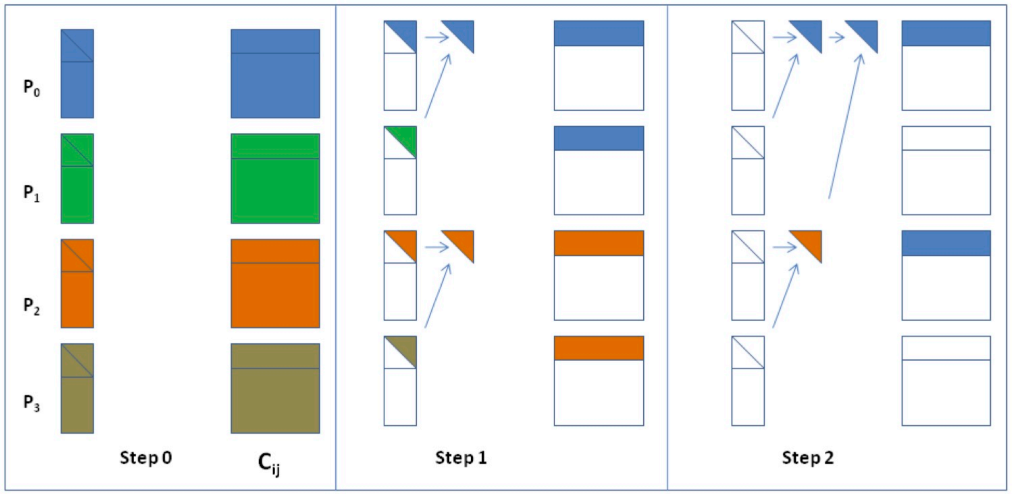
Algebra of TSQR

Parallel:

$$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$$

$$\begin{tikzcd}
W_0 \arrow[r] & R_{00} \arrow[dr] & & \\
W_1 \arrow[r] & R_{10} \arrow[r] & R_{01} \arrow[dr] & \\
W_2 \arrow[r] & R_{20} \arrow[r] & R_{11} \arrow[r] & R_{02} \\
W_3 \arrow[r] & R_{30} \arrow[ur] & &
\end{tikzcd}$$

CAQR



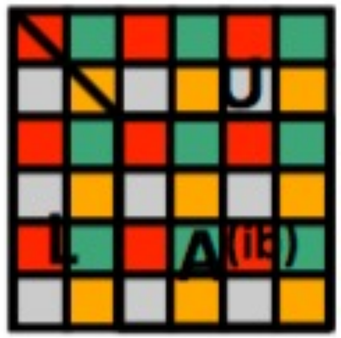
QR for General Matrices

- Cost of CAQR vs ScaLAPACK's PDGEQRF
  - n x n matrix on $P^{1/2}$ x $P^{1/2}$ processor grid, block size b
  - Flops: $(4/3)n^3/P + (3/4)n^2 b \log P/P^{1/2}$ vs $(4/3)n^3/P$
  - Bandwidth: $(3/4)n^2 \log P/P^{1/2}$ vs same
  - Latency: $2.5\, n \log P / \mathbf{b}$ vs $1.5\, n \log P$

- Close to optimal (modulo log P factors)
  - Assume: $O(n^2/P)$ memory/processor, $O(n^3)$ algorithm,
  - Choose b near $n / P^{1/2}$ (its upper bound)
  - Bandwidth lower bound:

    $\Omega(n^2 / P^{1/2})$ – just log(P) smaller
  - Latency lower bound:

    $\Omega(P^{1/2})$ – just polylog(P) smaller

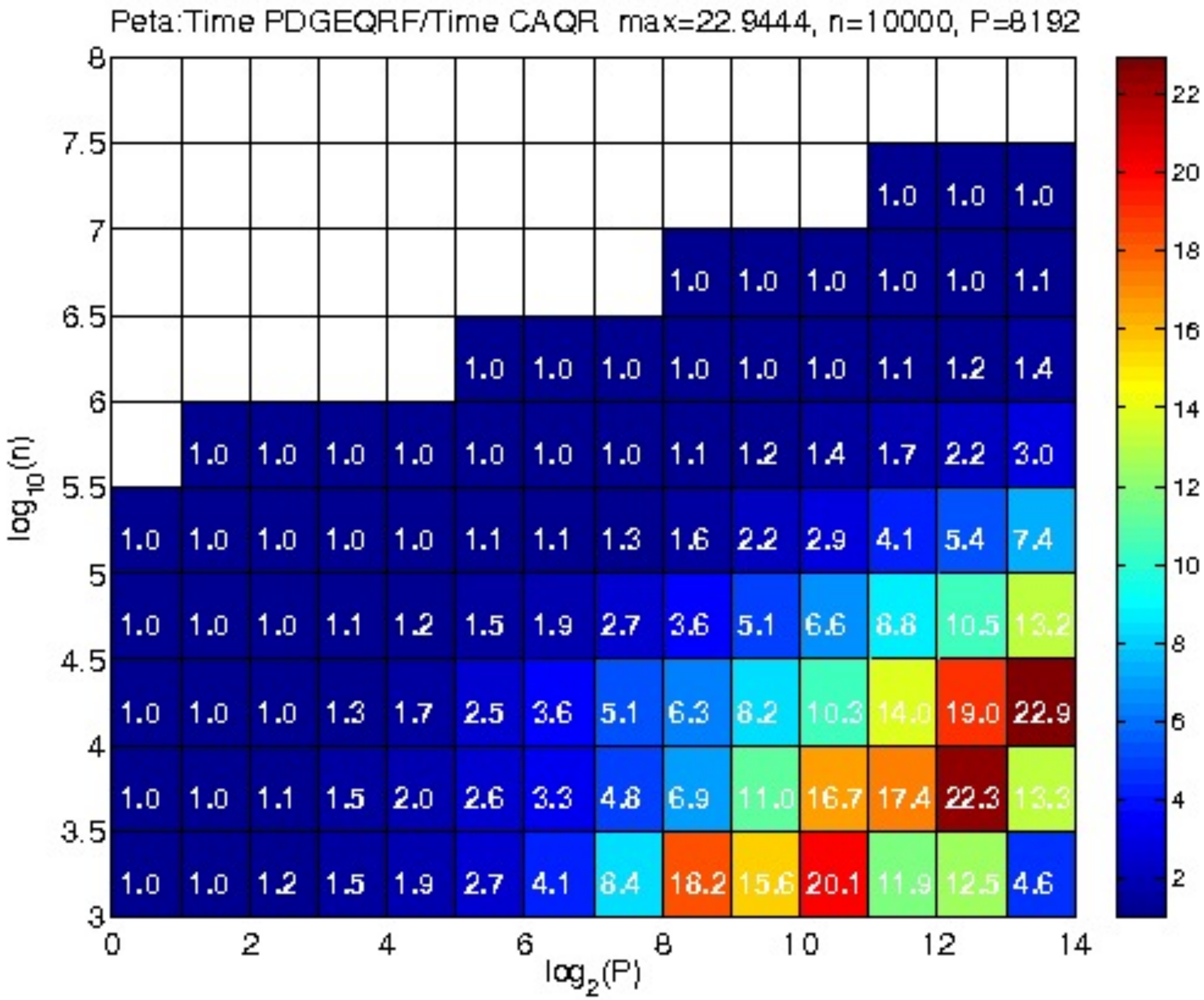


Performance of TSQR vs Sca/LAPACK

- Parallel
  - Intel Xeon (two socket, quad core machine), 2010
    - Up to **5.3x speedup** (8 cores, 10^5 x 200)
  - Pentium III cluster, Dolphin Interconnect, MPICH, 2008
    - Up to **6.7x speedup** (16 procs, 100K x 200)
  - BlueGene/L, 2008
    - Up to **4x speedup** (32 procs, 1M x 50)
  - Tesla C 2050 / Fermi (Anderson et al)
    - Up to **13x** (110,592 x 100)
  - Grid – **4x** on 4 cities vs 1 city (Dongarra, Langou et al)
  - QR computed locally using recursive algorithm (Elmroth-Gustavson) – enabled by TSQR

- Results from many papers, for some see [Demmel, LG, Hoemmen, Langou, SISC 12], [Donfack, LG, IPDPS 10].

Modeled Speedups of CAQR vs ScaLAPACK



Petascale

up to 22.9x

IBM Power 5

up to 9.7x

“Grid”

up to 11x

Petascale machine with 8192 procs, each at 500 GFlops/s, a bandwidth of 4 GB/s.

$$\gamma = 2\cdot 10^{-12} s, \alpha = 10^{-5} s, \beta = 2\cdot 10^{-9} s/word$$

Impact

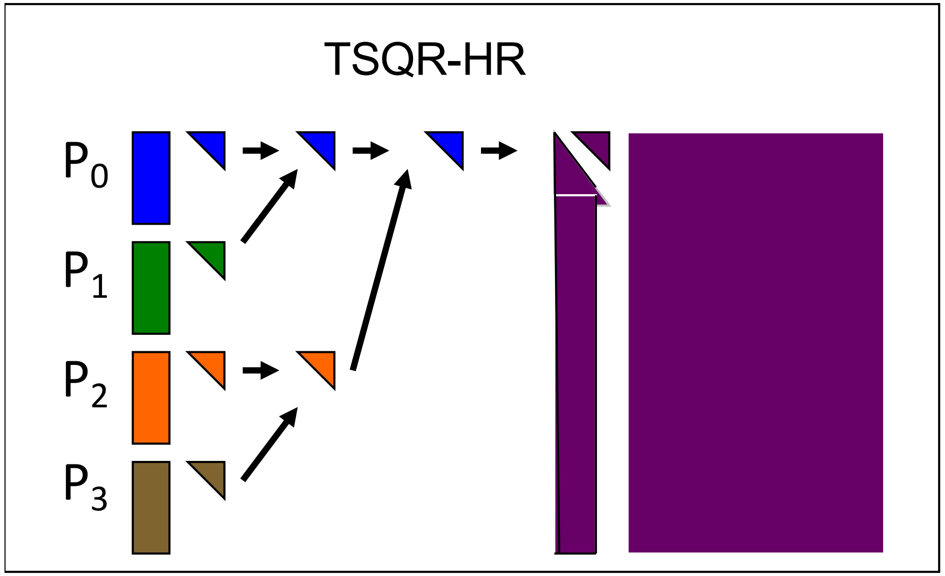
- TSQR/CAQR implemented in
  - Intel MKL library
  - GNU Scientific Library
  - ScaLAPACK
  - Spark for data mining

- CALU implemented in
  - Cray's libsci
  - To be implemented in lapack/scapalack

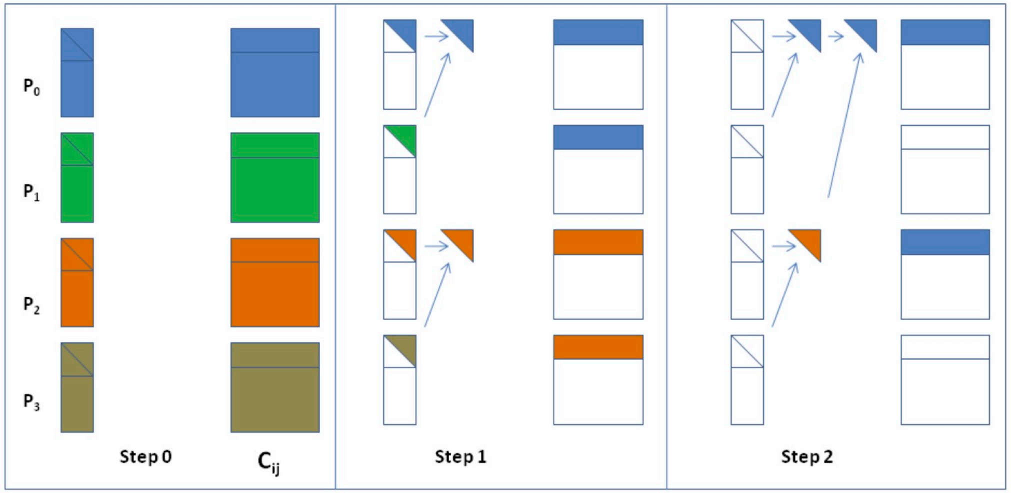
Algebra of TSQR

Parallel:

$$
W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}
\begin{array}{c} \to \\ \to \\ \to \\ \to \end{array}
\begin{array}{c} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{array}
\begin{array}{c} \searrow \\ \nearrow \\ \searrow \\ \nearrow \end{array}
\begin{array}{c} R_{01} \\ \\ R_{11} \end{array}
\begin{array}{c} \searrow \\ \nearrow \end{array}
R_{02}
$$



CAQR



Reconstruct Householder vectors from TSQR

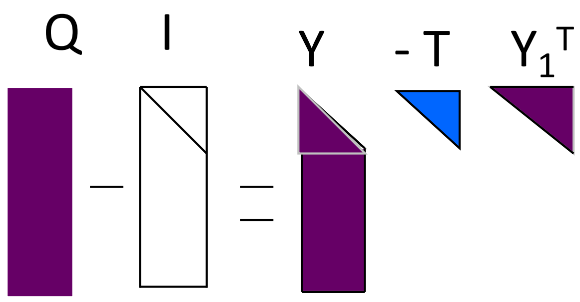
The QR factorization using Householder vectors

$$W = QR = (I - YTY_1^T)R$$

can be re-written as an LU factorization

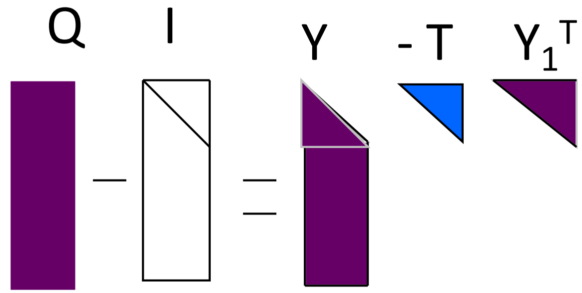
$$W - R = Y(-TY_1^T)R$$

$$Q - I = Y(-TY_1^T)$$



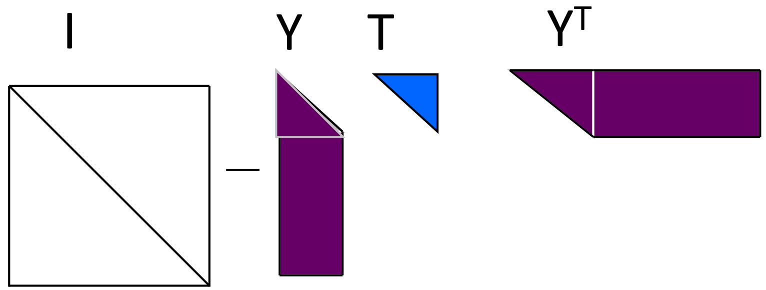
Reconstruct Householder vectors TSQR-HR

1. Perform TSQR
2. Form Q explicitly (tall-skinny orthonormal factor)
3. Perform LU decomposition: $Q - I = LU$

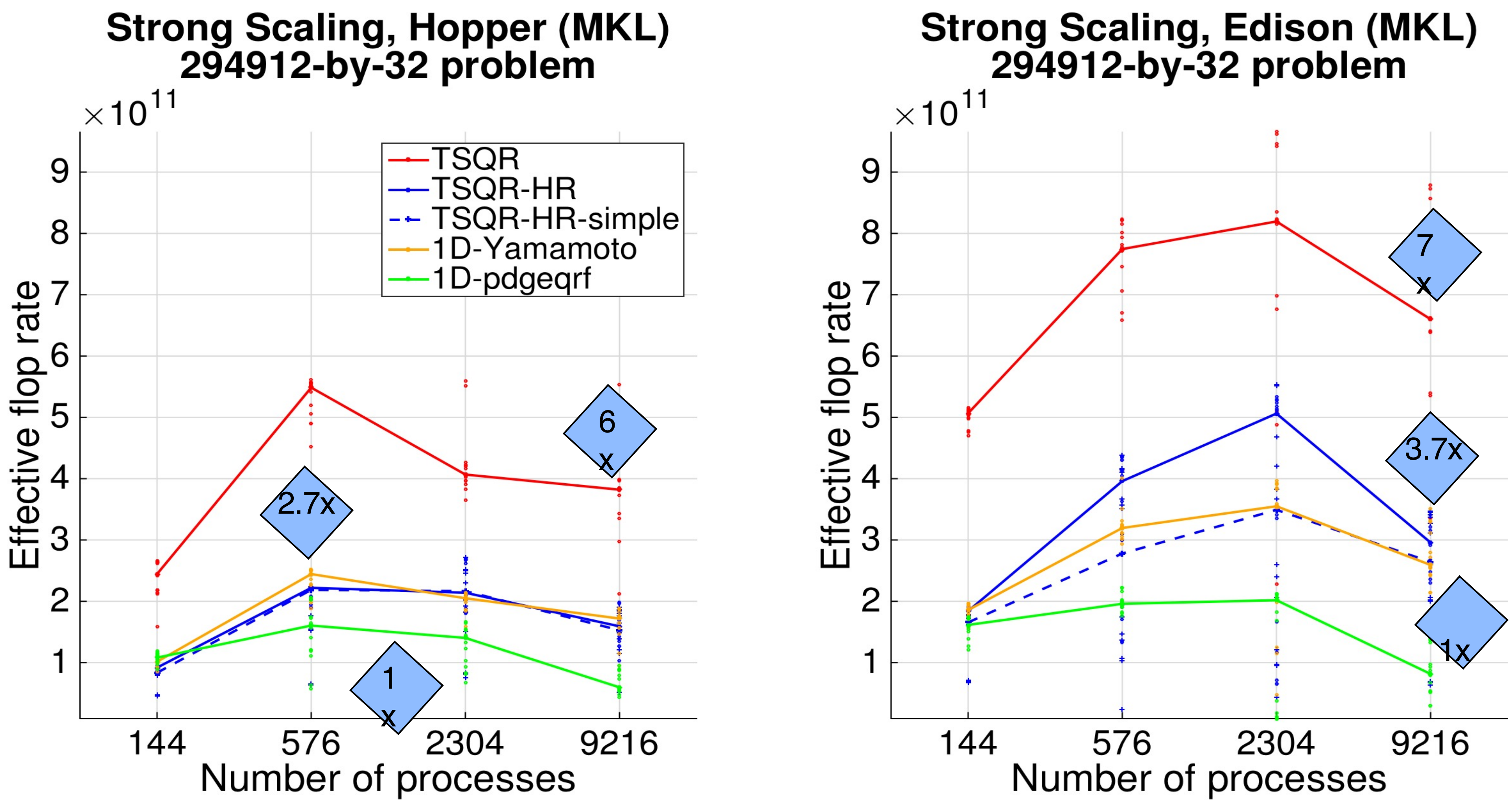


4. Set $Y = L$
5. Set $T = -U\,Y_1^{-T}$

$$I - YTY^T = I - \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} T \begin{bmatrix} Y_1^T & Y_2^T \end{bmatrix}$$



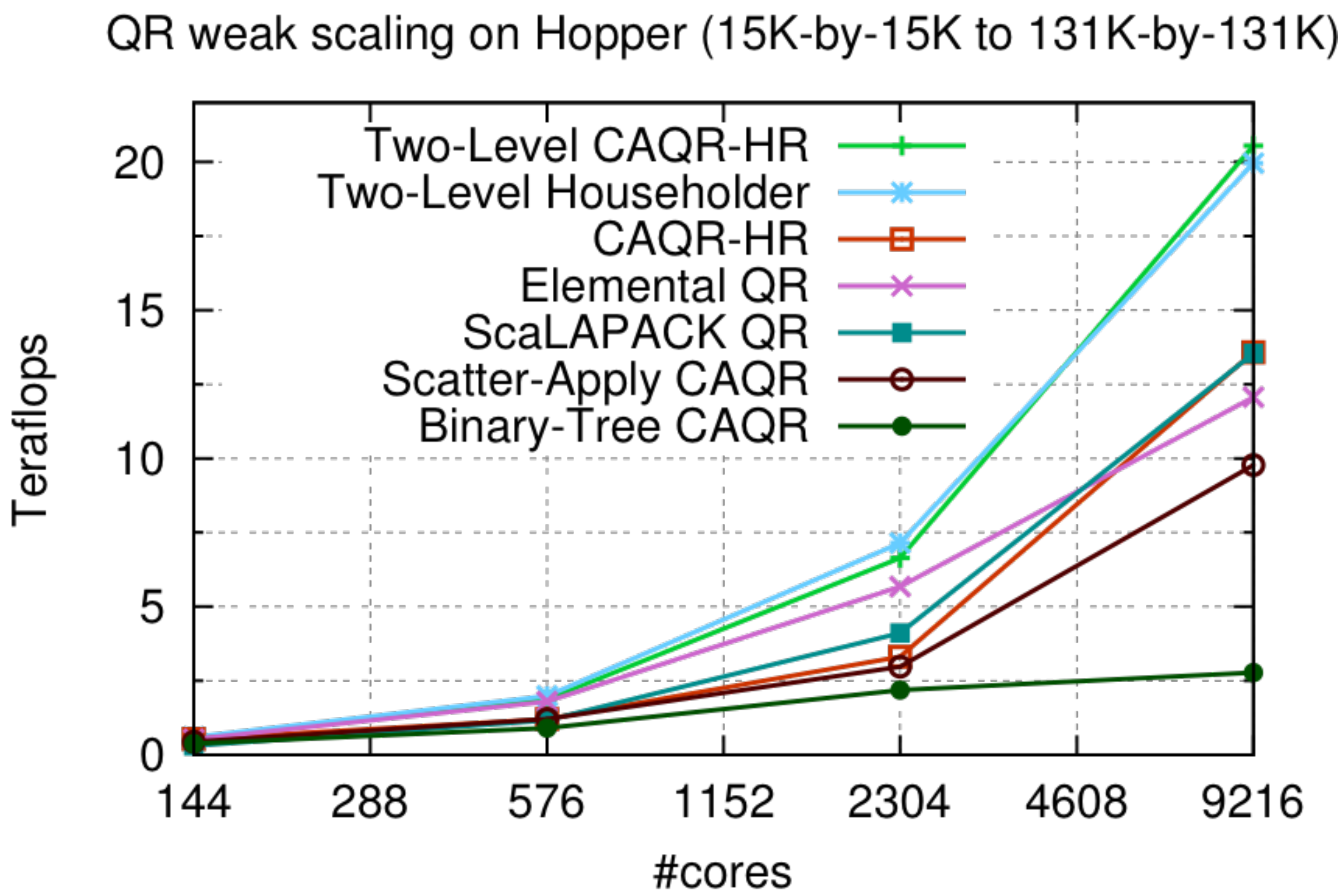
Strong scaling



- Hopper: Cray XE6 (NERSC) – 2 x 12-core AMD Magny-Cours (2.1 GHz)
- Edison: Cray CX30 (NERSC) – 2 x 12-core Intel Ivy Bridge (2.4 GHz)
- Effective flop rate, computed by dividing $2mn^2 - 2n^3/3$ by measured runtime

Ballard, Demmel, LG, Jacquelin, Knight, Nguyen, and Solomonik, 2015.

Weak scaling QR on Hopper



- Matrix of size 15K-by-15K to 131K-by-131K
- Hopper: Cray XE6 supercomputer (NERSC) – dual socket 12-core Magny-Cours Opteron (2.1 GHz)

The LU factorization of a tall skinny matrix

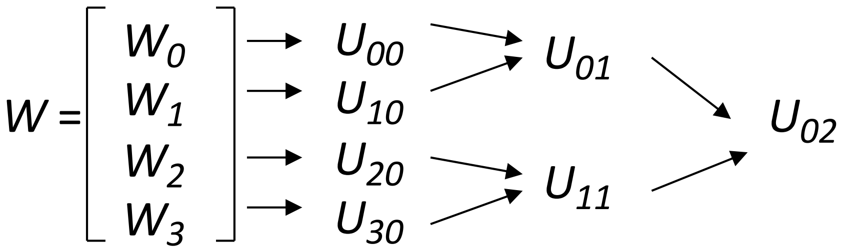
First try the obvious generalization of TSQR.

$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \underbrace{\begin{pmatrix} \Pi_{00} & & & \\ & \Pi_{10} & & \\ & & \Pi_{20} & \\ & & & \Pi_{30} \end{pmatrix}}_{\Pi_0} \cdot \begin{pmatrix} L_{00} & & & \\ & L_{10} & & \\ & & L_{20} & \\ & & & L_{30} \end{pmatrix} \cdot \begin{pmatrix} U_{00} \\ U_{10} \\ U_{20} \\ U_{30} \end{pmatrix}$$

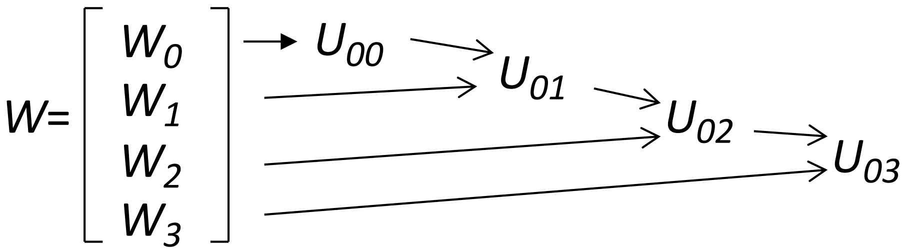
$$\begin{pmatrix} U_{00} \\ U_{10} \\ U_{20} \\ U_{30} \end{pmatrix} = \underbrace{\begin{pmatrix} \Pi_{01} & \\ & \Pi_{11} \end{pmatrix}}_{\Pi_1} \cdot \begin{pmatrix} L_{01} & \\ & L_{11} \end{pmatrix} \cdot \begin{pmatrix} U_{01} \\ U_{11} \end{pmatrix}$$

Obvious generalization of TSQR to LU

- Block parallel pivoting:
  - uses a binary tree and is optimal in the parallel case



- Block pairwise pivoting:
  - uses a flat tree and is optimal in the sequential case
  - introduced by Barron and Swinnerton-Dyer, 1960: block LU factorization used to solve a system with 100 equations on EDSAC 2 computer using an auxiliary magnetic-tape
  - used in PLASMA for multicore architectures and FLAME for out-of-core algorithms and for multicore architectures



Stability of the LU factorization

- The backward stability of the LU factorization of a matrix A of size n-by-n

$$\left\| |\hat{L}| \cdot |\hat{U}| \right\|_\infty \leq (1 + 2(n^2 - n) g_w) \|A\|_\infty$$

depends on the growth factor

$$g_W = \frac{\max_{i,j,k} |a_{ij}^k|}{\max_{i,j} |a_{ij}|}$$ where a_{ij}^k are the values at the k-th step.

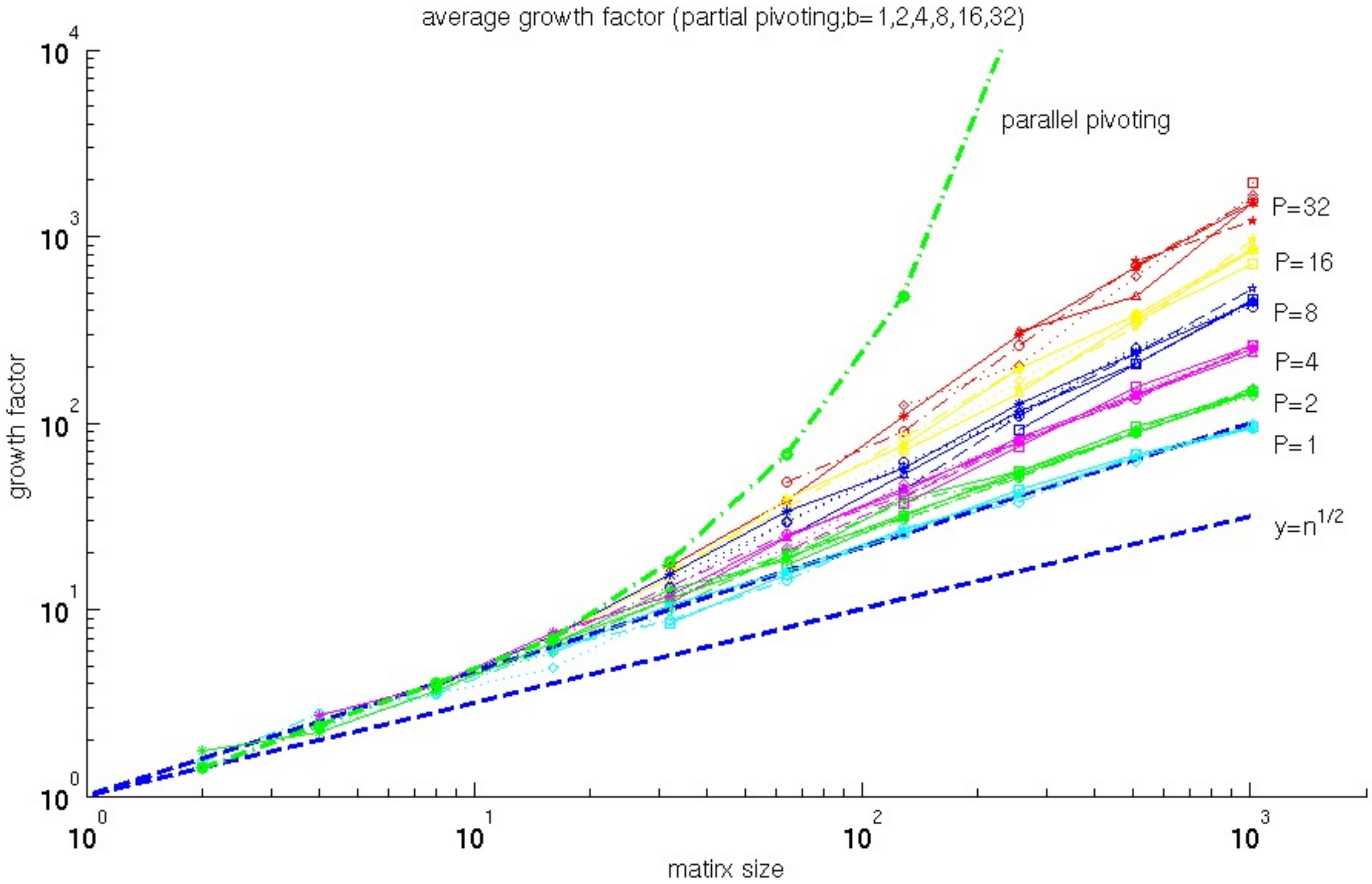
- $g_W \leq 2^{n-1}$, attained for Wilkinson matrix

  but in practice it is on the order of $n^{2/3}$ -- $n^{1/2}$

- Two reasons considered to be important for the average case stability [Trefethen and Schreiber, 90] :

- the multipliers in L are small,
- the correction introduced at each elimination step is of rank 1.

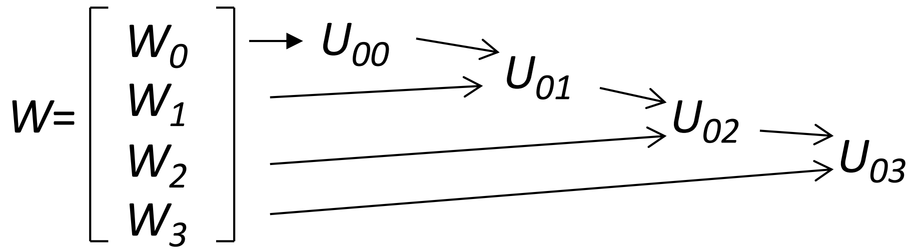
Block parallel pivoting

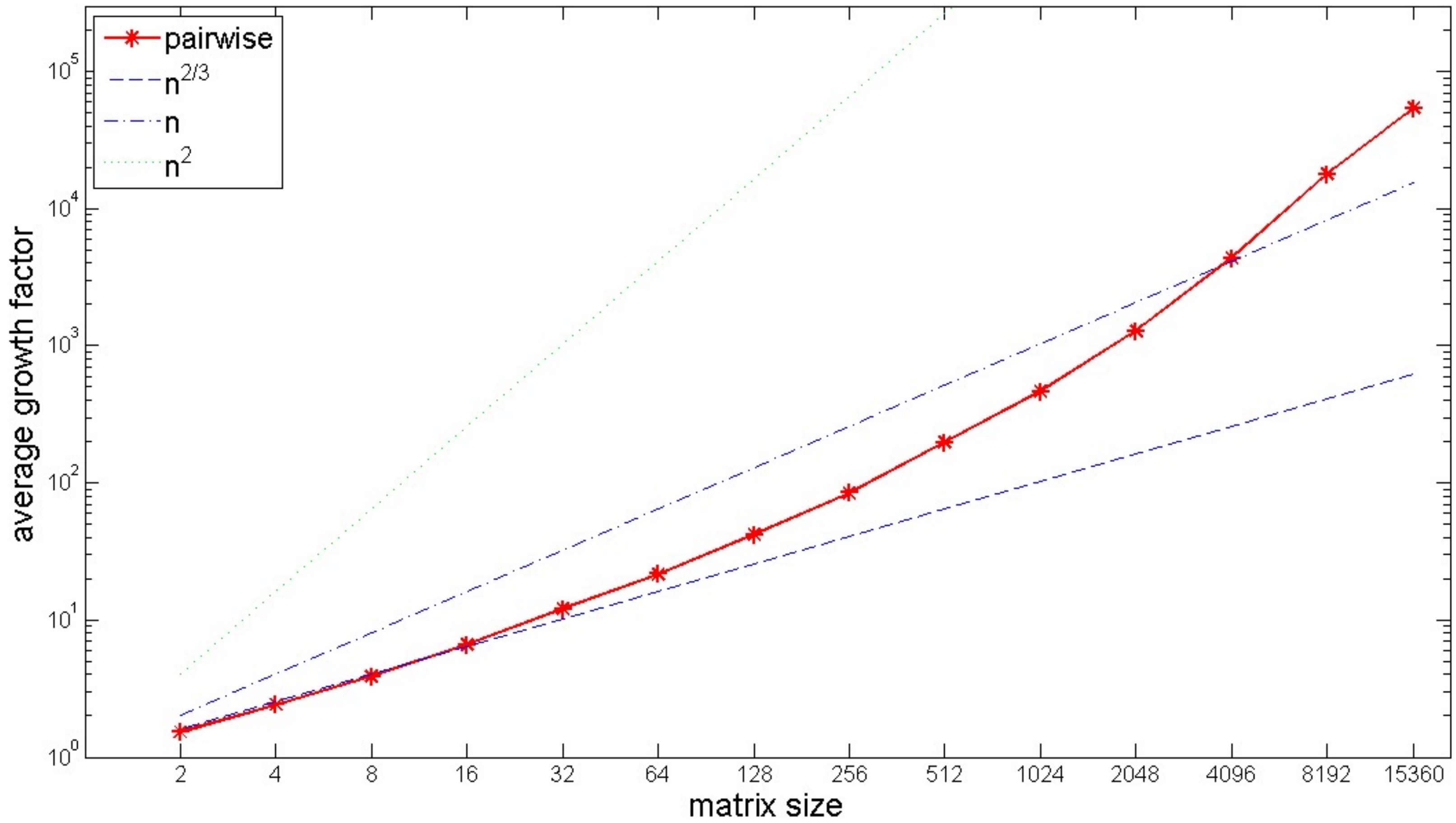


- Unstable for large number of processors P
- When P=number rows, it corresponds to parallel pivoting, known to be unstable (Trefethen and Schreiber, 90)

Block pairwise pivoting

- Results shown for random matrices
- Will become unstable for large matrices





Tournament pivoting - the overall idea

- At each iteration of a block algorithm

$$A = \begin{pmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{pmatrix} \begin{matrix} \} & b \\ \} & n-b \end{matrix} \text{, where } \quad W = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

  - Preprocess W to find at low communication cost good pivots for the LU factorization of W, return a permutation matrix P.
  - Permute the pivots to top, ie compute PA.
  - Compute LU with no pivoting of W, update trailing matrix.

$$PA = \begin{pmatrix} L_{11} & \\ L_{21} & I_{n-b} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & A_{22} - L_{21}U_{12} \end{pmatrix}$$

Tournament pivoting for a tall skinny matrix

1) Compute GEPP factorization of each W_i, find permutation Π_0

$$W = \begin{pmatrix} W_0 \\ \hline W_1 \\ \hline W_2 \\ \hline W_3 \end{pmatrix} = \begin{pmatrix} \Pi_{00} L_{00} U_{00} \\ \hline \Pi_{10} L_{10} U_{10} \\ \hline \Pi_{20} L_{20} U_{20} \\ \hline \Pi_{30} L_{30} U_{30} \end{pmatrix},$$

Pick b pivot rows, form A_{00}

Same for A_{10}

Same for A_{20}

Same for A_{30}

2) Perform $\log_2(P)$ times GEPP factorizations of 2b-by-b rows, find permutations Π_1, Π_2

$$\begin{pmatrix} A_{00} \\ A_{10} \\ \hline A_{20} \\ A_{30} \end{pmatrix} = \begin{pmatrix} \Pi_{01} L_{01} U_{01} \\ \hline \Pi_{11} L_{11} U_{11} \end{pmatrix}$$

Pick b pivot rows, form A_{01}

Same for A_{11}

3) Compute LU factorization with no pivoting of the permuted matrix:

$$\Pi_2^T \Pi_1^T \Pi_0^T W = LU$$

Tournament pivoting

P_0: $$W_0 = \begin{pmatrix} 2 & 4 \\ 0 & 1 \\ 2 & 0 \\ 1 & 2 \end{pmatrix} = \Pi_0 L_0 U_0 \qquad \Pi_0^T W_0 = \begin{pmatrix} 2 & 4 \\ 2 & 0 \end{pmatrix}$$

P_1: $$W_1 = \begin{pmatrix} 2 & 0 \\ 0 & 0 \\ 4 & 1 \\ 1 & 0 \end{pmatrix} = \Pi_1 L_1 U_1 \qquad \Pi_1^T W_1 = \begin{pmatrix} 4 & 1 \\ 2 & 0 \end{pmatrix}$$

P_2: $$W_2 = \begin{pmatrix} 0 & 1 \\ 1 & 4 \\ 0 & 0 \\ 0 & 2 \end{pmatrix} = \Pi_2 L_2 U_2 \qquad \Pi_2^T W_2 = \begin{pmatrix} 1 & 4 \\ 0 & 2 \end{pmatrix}$$

P_3: $$W_3 = \begin{pmatrix} 2 & 1 \\ 0 & 2 \\ 1 & 0 \\ 4 & 2 \end{pmatrix} = \Pi_3 L_3 U_3 \qquad \Pi_3^T W_3 = \begin{pmatrix} 4 & 2 \\ 0 & 2 \end{pmatrix}$$

$$\overline{W_0} = \begin{pmatrix} 2 & 4 \\ 2 & 0 \\ 4 & 1 \\ 2 & 0 \end{pmatrix} = \overline{\Pi}_0 \overline{L}_0 \overline{U}_0 \qquad \overline{\Pi}_0^T \overline{W}_0 = \begin{pmatrix} 4 & 1 \\ 2 & 4 \end{pmatrix}$$

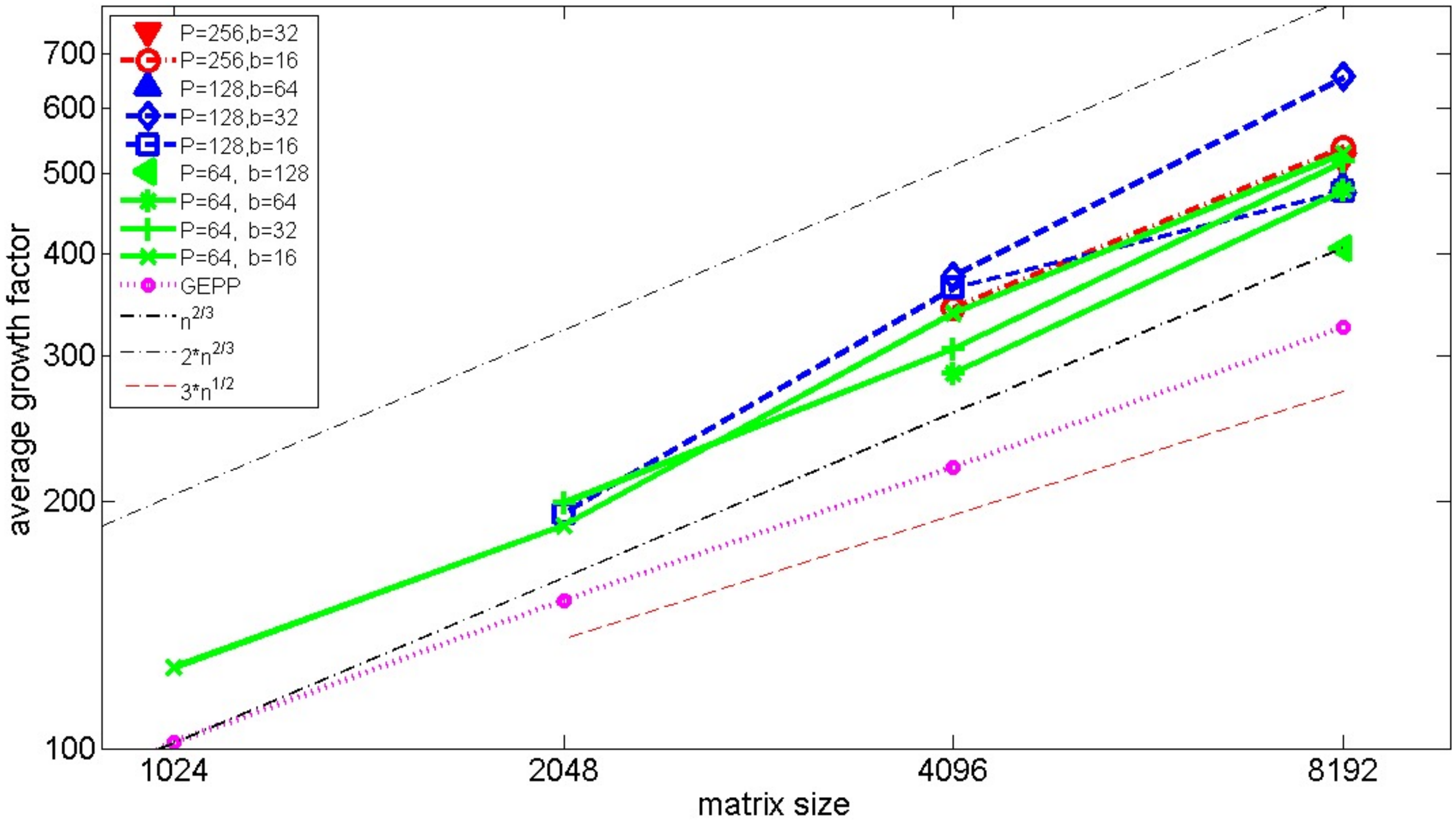
$$\overline{W_2} = \begin{pmatrix} 1 & 4 \\ 0 & 2 \\ 4 & 2 \\ 0 & 2 \end{pmatrix} = \overline{\Pi}_2 \overline{L}_2 \overline{U}_2 \qquad \overline{\Pi}_2^T \overline{W}_2 = \begin{pmatrix} 4 & 2 \\ 1 & 4 \end{pmatrix}$$

$$\underline{W}_0 = \begin{pmatrix} 4 & 1 \\ 2 & 4 \\ 4 & 2 \\ 1 & 4 \end{pmatrix} = \underline{\Pi}_0 \underline{L}_0 \underline{U}_0 \qquad \underline{\Pi}_0^T \underline{W}_0 = \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}$$

Good pivots for factorizing W

time $\longrightarrow$

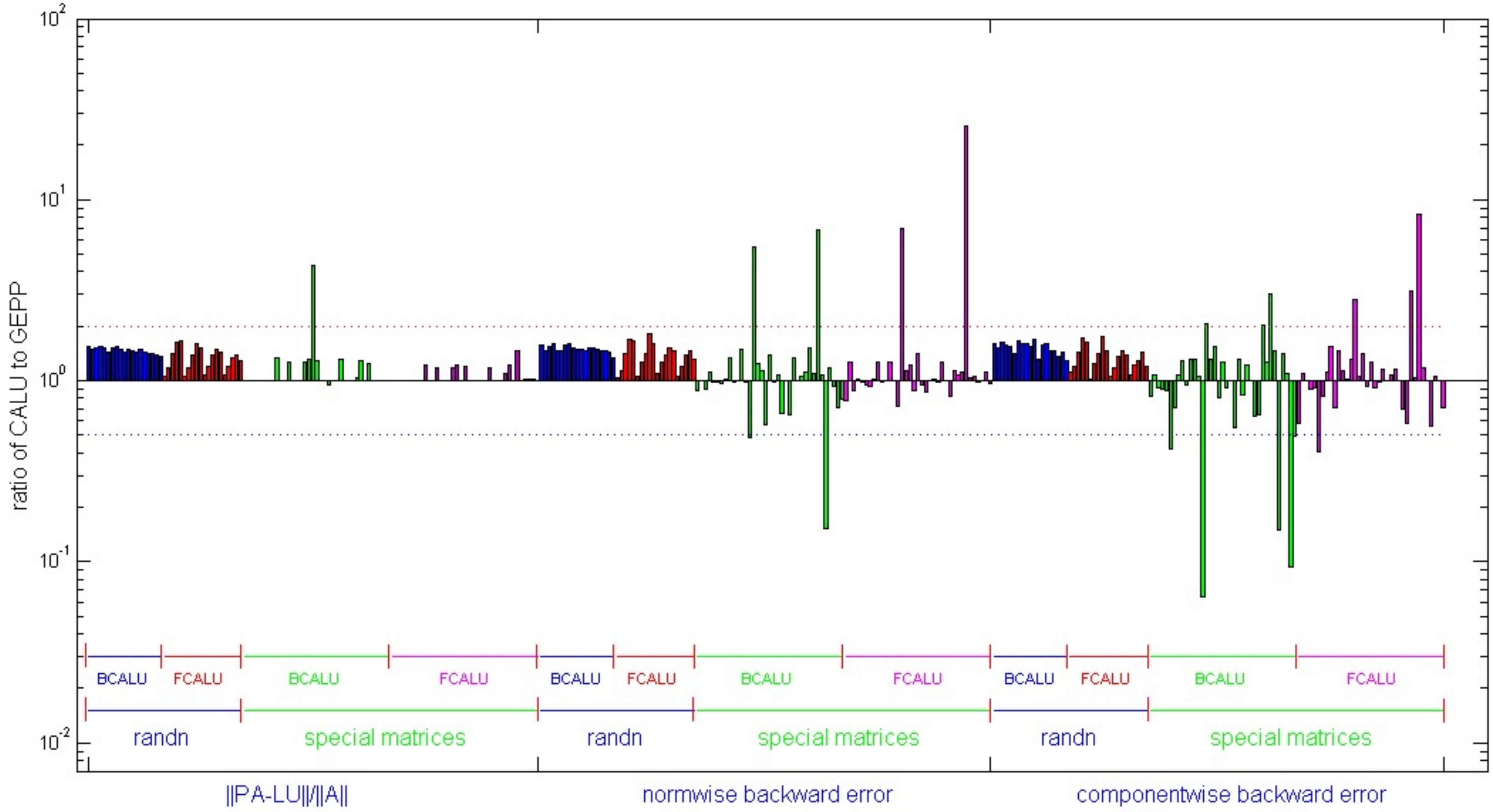
Growth factor for binary tree based CALU



- Random matrices from a normal distribution
- Same behaviour for all matrices in our test, and |L| <= 4.2

Stability of CALU (experimental results)

- Results show ||PA-LU||/||A||, normwise and componentwise backward errors, for random matrices and special ones
  - See [LG, Demmel, Xiang, SIMAX 2011] for details
  - BCALU denotes binary tree based CALU and FCALU denotes flat tree based CALU



Our "proof of stability" for CALU

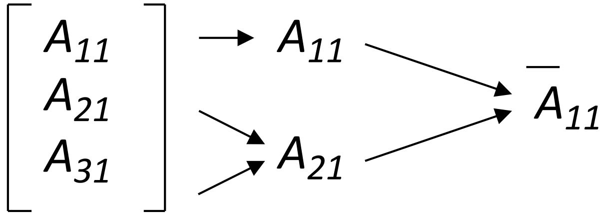
- CALU as stable as GEPP in following sense:

  In exact arithmetic, CALU process on a matrix A is equivalent to GEPP process on a larger matrix G whose entries are blocks of A and zeros.

- Example of one step of tournament pivoting:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{pmatrix}$$

tournament pivoting:



$$G = \begin{pmatrix} \bar{A}_{11} & & \bar{A}_{12} \\ A_{21} & A_{21} & \\ & -A_{31} & A_{32} \end{pmatrix}$$

- Proof possible by using original rows of A during tournament pivoting (not the computed rows of U).

Growth factor in exact arithmetic

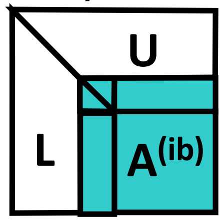
- Matrix of size m-by-n, reduction tree of height H=log(P).
- (CA)LU_PRRP select pivots using strong rank revealing QR (A. Khabou, J. Demmel, LG, M. Gu, SIMAX 2013)
- "In practice" means observed/expected/conjectured values.

	CALU	GEPP
Upper bound	$2^{n(\log(P)+1)-1}$	2^{n-1}
In practice	$n^{2/3}$ -- $n^{1/2}$	$n^{2/3}$ -- $n^{1/2}$

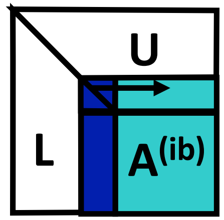
Better bounds

CALU – a communication avoiding LU factorization

- Consider a 2D grid of P processors P_r-by-P_c , using a 2D block cyclic layout with square blocks of size b.



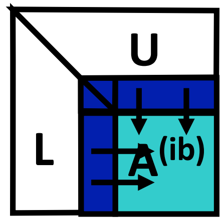
For ib = 1 to n-1 step b
  $A^{(ib)} = A(ib:n, ib:n)$



(1) Find permutation for current panel using TSLU $O(n/b \log_2 P_r)$

(2) Apply all row permutations (pdlaswp) $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast pivot information along the rows of the grid

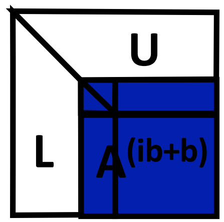


(3) Compute panel factorization (dtrsm)

$O(n/b \log_2 P_c)$

(4) Compute block row of U (pdtrsm)

- broadcast right diagonal part of L of current panel



(5) Update trailing matrix (pdgemm) $O(n/b(\log_2 P_c + \log_2 P_r))$

- broadcast right block column of L
- broadcast down block row of U

LU for General Matrices

- Cost of CALU vs ScaLAPACK's PDGETRF
  - n x n matrix on $P^{1/2}$ x $P^{1/2}$ processor grid, block size b
  - Flops: $(2/3)n^3/P + (3/2)n^2b / P^{1/2}$ vs $(2/3)n^3/P + n^2b/P^{1/2}$
  - Bandwidth: $n^2 \log P/P^{1/2}$ vs same
  - Latency: $3 n \log P / \mathbf{b}$ vs $1.5 n \log P + 3.5n \log P / b$
- Close to optimal (modulo log P factors)
  - Assume: $O(n^2/P)$ memory/processor, $O(n^3)$ algorithm,
  - Choose b near $n / P^{1/2}$ (its upper bound)
  - Bandwidth lower bound:
    $\Omega(n^2 /P^{1/2})$ – just log(P) smaller
  - Latency lower bound:
    $\Omega(P^{1/2})$ – just polylog(P) smaller

Performance vs ScaLAPACK

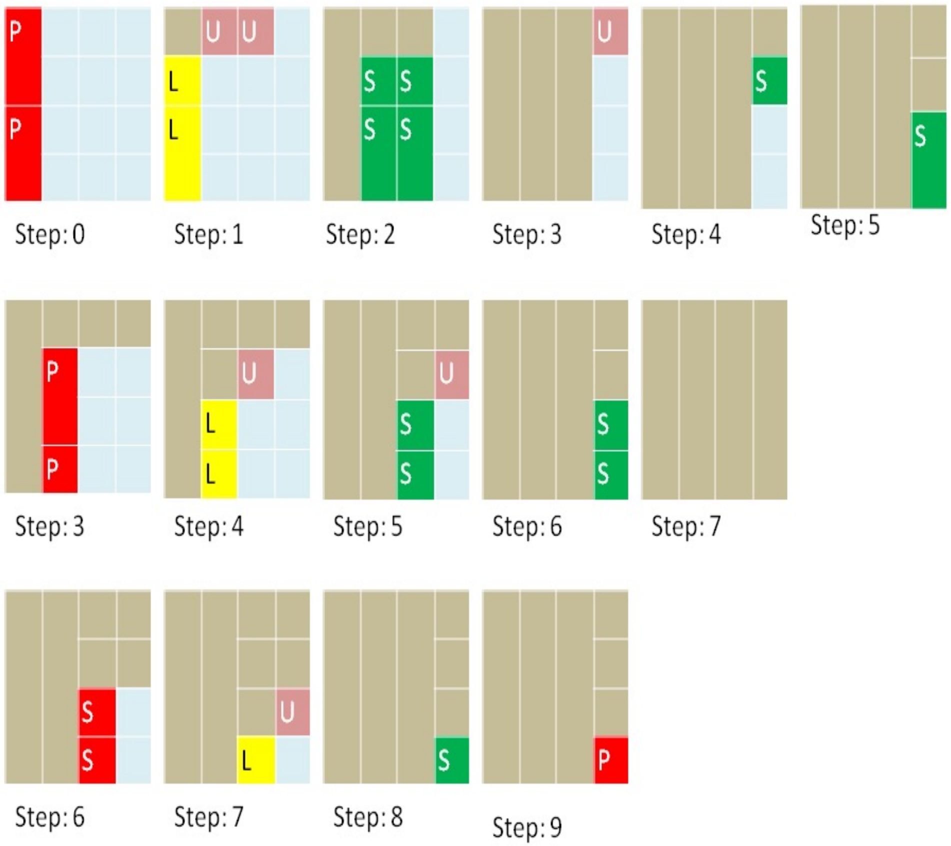
- Parallel TSLU (LU on tall-skinny matrix)
  - IBM Power 5
    - Up to **4.37x** faster (16 procs, 1M x 150)
  - Cray XT4
    - Up to **5.52x** faster (8 procs, 1M x 150)

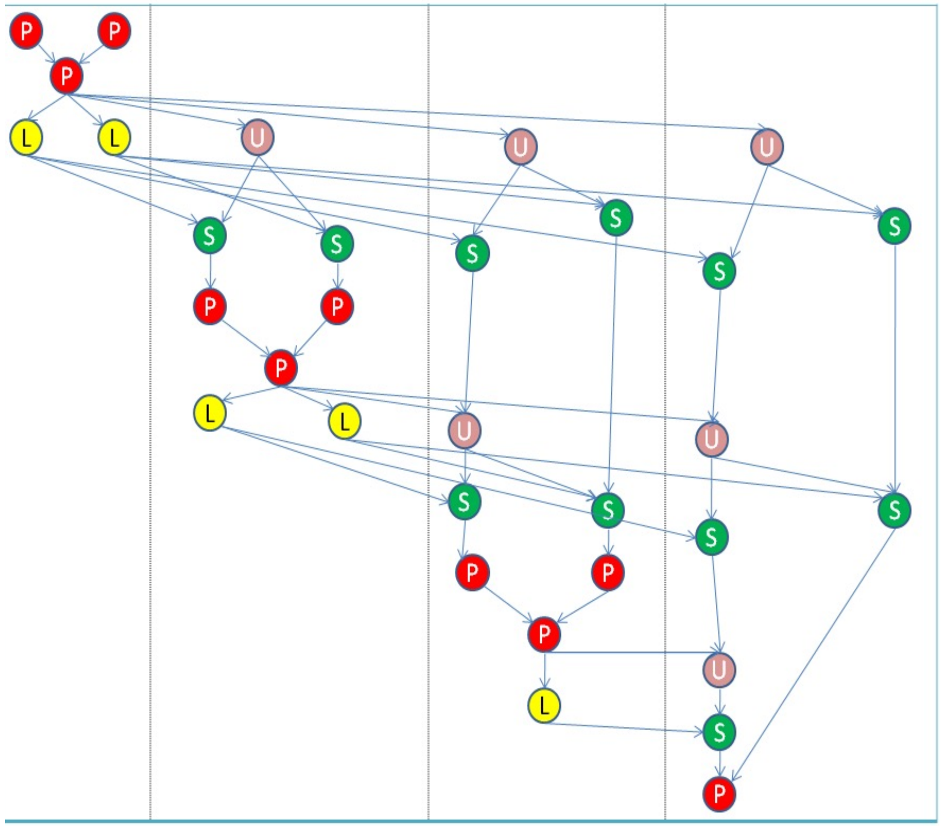
- Parallel CALU (LU on general matrices)
  - Intel Xeon (two socket, quad core)
    - Up to **2.3x** faster (8 cores, 10^6 x 500)
  - IBM Power 5
    - Up to **2.29x** faster (64 procs, 1000 x 1000)
  - Cray XT4
    - Up to **1.81x** faster (64 procs, 1000 x 1000)

- Details in SC08 (LG, Demmel, Xiang), IPDPS'10 (S. Donfack, LG).

CALU and its task dependency graph

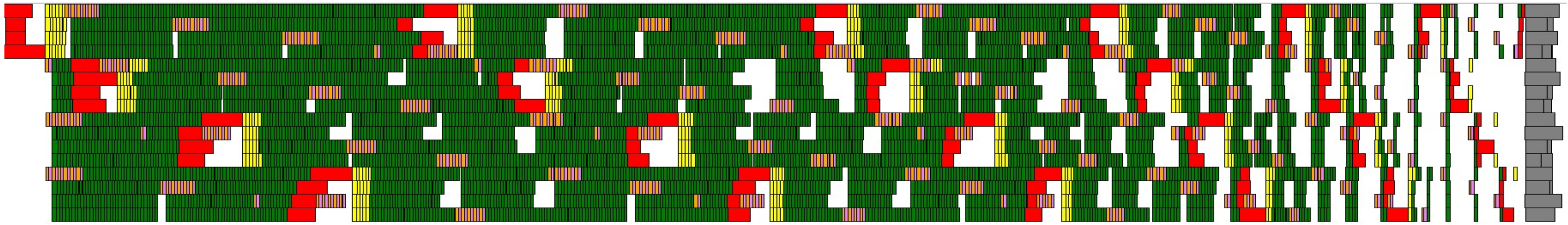
- The matrix is partitioned into blocks of size T x b.
- The computation of each block is associated with a task.



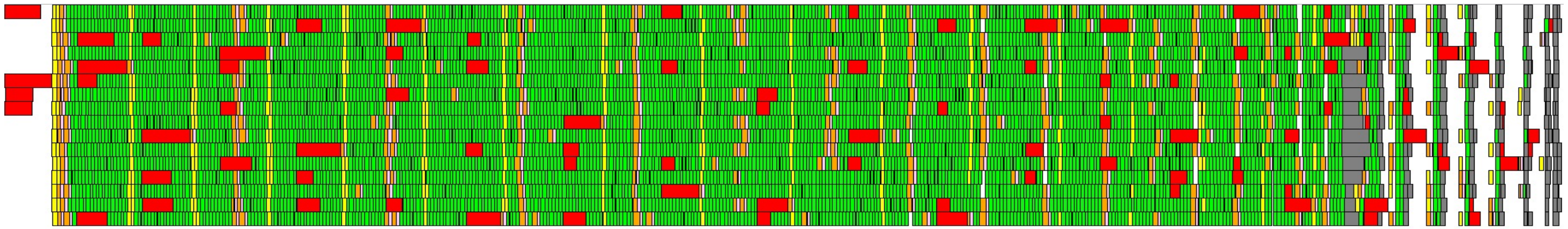


Scheduling CALU's Task Dependency Graph

- Static scheduling
  - \+ Good locality of data
  - \- Ignores noise



- Dynamic scheduling
  - \+ Keeps cores busy
  - \- Poor usage of data locality
  - \- Can have large dequeue overhead



Lightweight scheduling

- Emerging complexities of multi- and mani-core processors suggest a need for self-adaptive strategies
  - One example is work stealing
- Goal:
  - Design a tunable strategy that is able to provide a good trade-off between load balance, data locality, and dequeue overhead.
  - Provide performance consistency
- Approach: combine static and dynamic scheduling
  - Shown to be efficient for regular mesh computation [B. Gropp and V. Kale]

	Design space		
Data layout/scheduling	Static	Dynamic	Static/(%dynamic)
Column Major Layout (CM)		√	
Block Cyclic Layout (BCL)	√	√	√
2-level Block Layout (2l-BL)	√	√	√

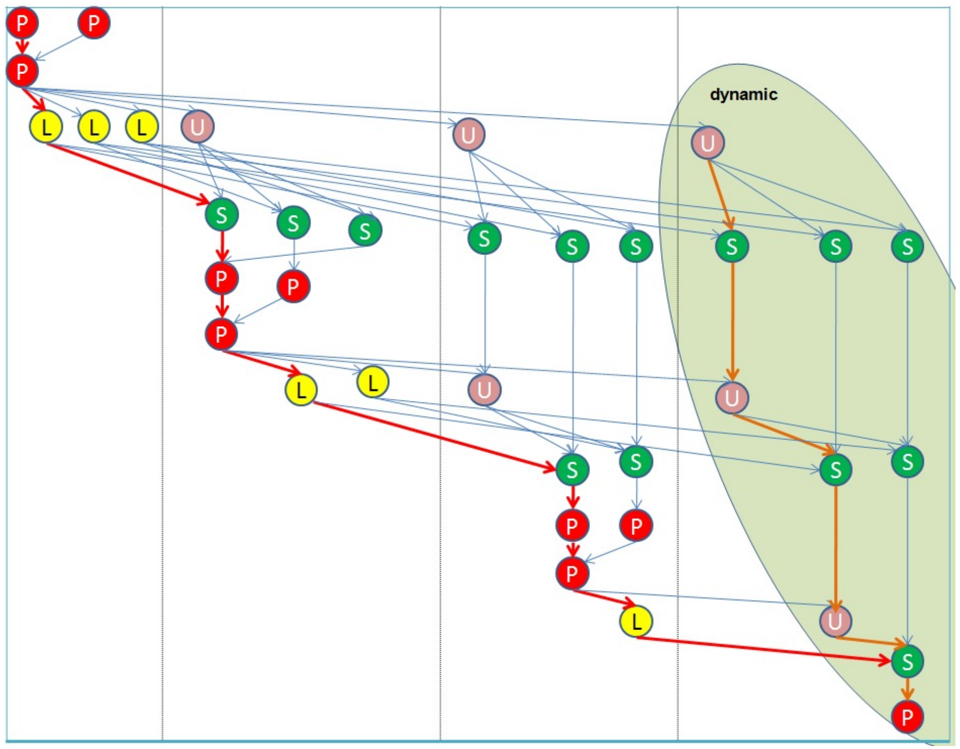
S. Donfack, LG, B. Gropp, V. Kale,IPDPS 2012

Lightweight scheduling

- A self-adaptive strategy to provide
  - A good trade-off between load balance, data locality, and dequeue overhead.
  - Performance consistency
  - Shown to be efficient for regular mesh computation [B. Gropp and V. Kale]

Combined static/dynamic scheduling:

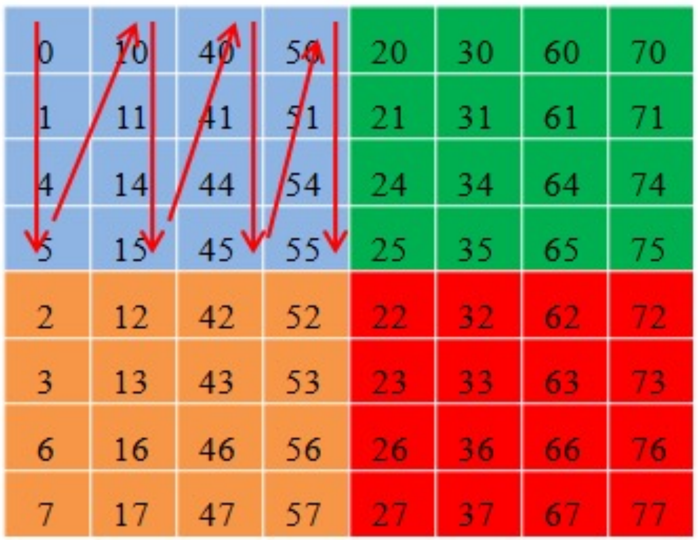
- A thread executes in priority its statically assigned tasks
- When no task ready, it picks a ready task from the dynamic part
- The size of the dynamic part is guided by a performance model



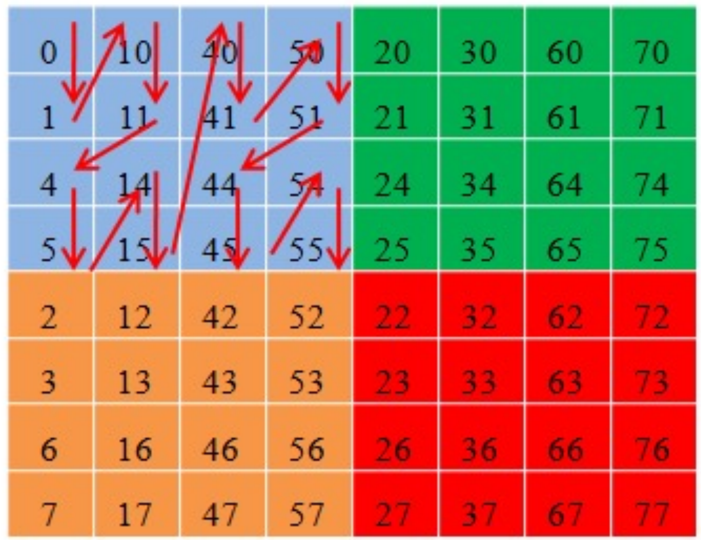
S. Donfack, LG, B. Gropp, V. Kale, 2012

Data layout and other optimizations

- Three data distributions investigated
  - CM : Column major order for the entire matrix
  - BCL : Each thread stores contiguously (CM) the data on which it operates
  - 2l-BL : Each thread stores in blocks the data on which it operates



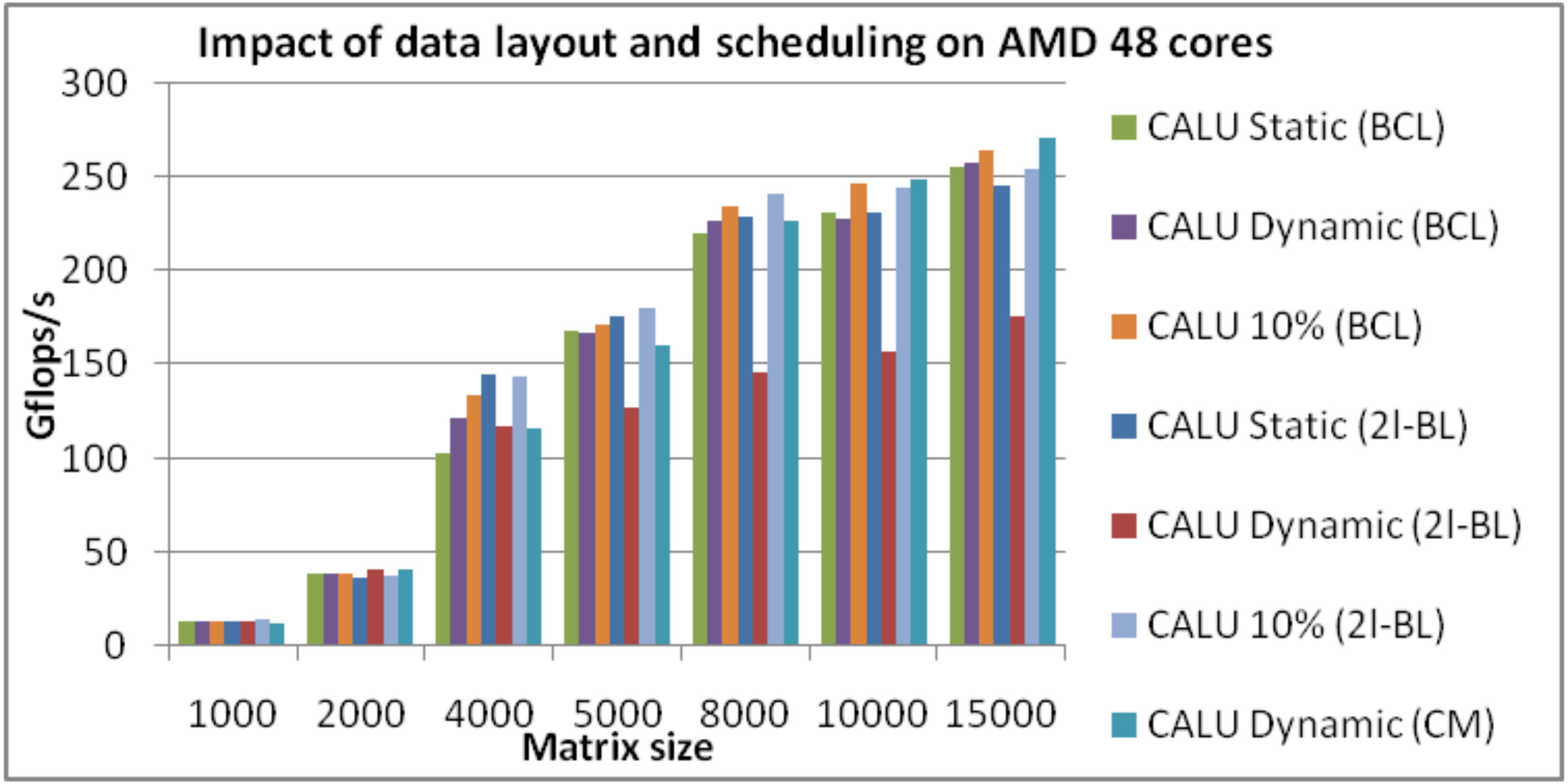
Block cyclic layout (BCL)



Two level block layout (2l-BL)

- And other optimizations
  - Updates (dgemm) performed on several blocks of columns (for BCL and CM layouts)

Impact of data layout

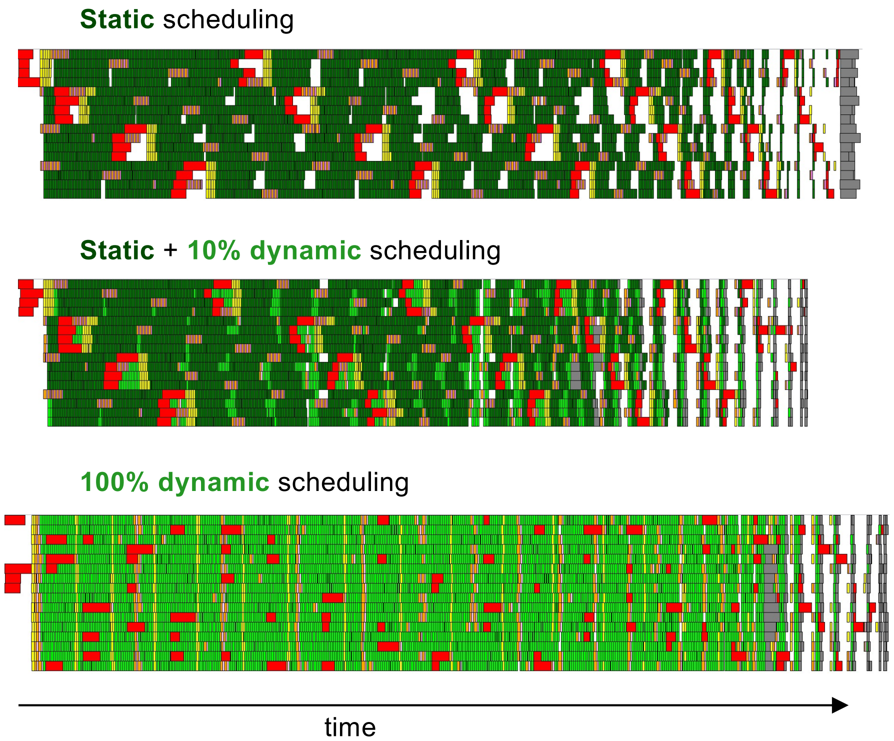


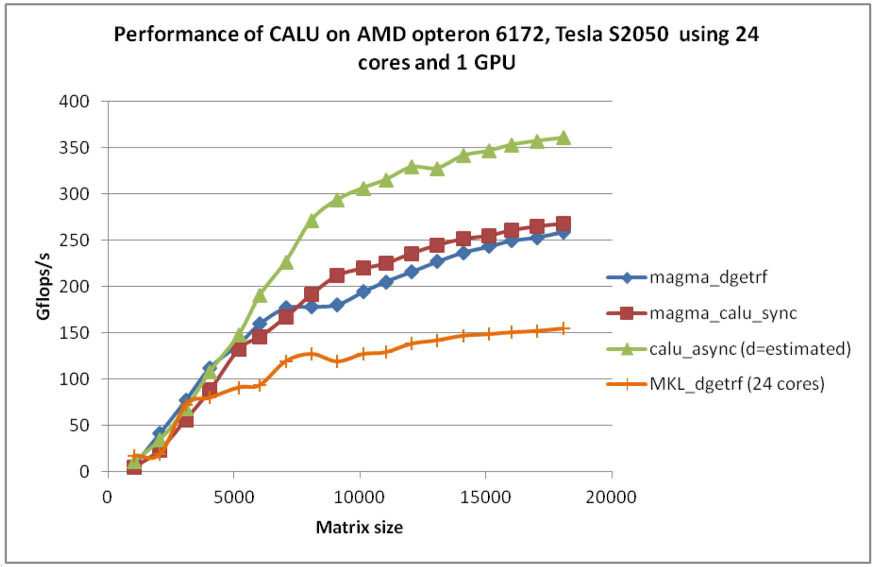
Eight socket, six core machine based on AMD Opteron processor (U. of Tennessee).

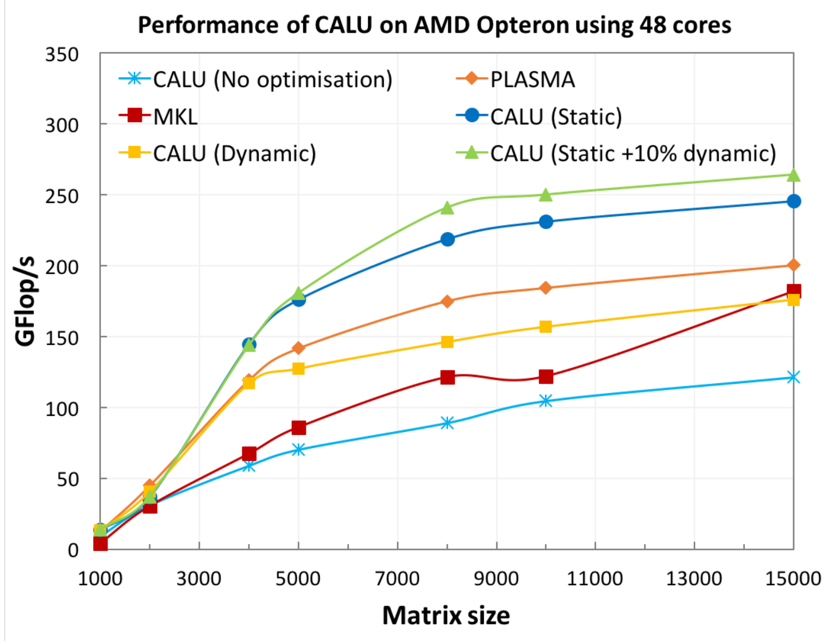
BCL : Each thread stores contiguously (CM) its data

2l-BL : Each thread stores in blocks its data

Best performance of CALU on multicore architectures







- Reported performance for PLASMA uses LU with block pairwise pivoting.
- GPU data courtesy of S. Donfack