Lecture 7: Dense Linear Algebra

Outline

- Dense Linear Algebra Overview
- Lower Bounds on Communication
- Parallel Matrix Multiply
- LU, QR factorizations

Dense Linear Algebra Overview

What is dense linear algebra?

- Not just matmul!
- Linear Systems: Ax=b
- Least Squares: choose x to minimize $||Ax-b||_2$
  - Overdetermined or underdetermined
  - Unconstrained, constrained, weighted
- Eigenvalues and vectors of Symmetric Matrices
  - Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)
- Eigenvalues and vectors of Unsymmetric matrices
  - Eigenvalues, Schur form, eigenvectors, invariant subspaces
  - Standard, Generalized
- Singular Values and vectors (SVD)
  - Standard, Generalized
- Different matrix structures
  - Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded ...
- Level of detail
  - Simple Driver ("x=A\b")
  - Expert Drivers with error bounds, extra-precision, other options
  - Lower level routines ("apply certain kind of orthogonal transformation", matmul...)

A brief history of (Dense) Linear Algebra software

- Mid 60's
  - Libraries like EISPACK (for eigenvalue problems)

- Then the BLAS (1) were invented (1973-1977)
  - Standard library of 15 operations (mostly) on vectors
    - "AXPY" ($y = \alpha \cdot x + y$), dot product, scale ($x = \alpha \cdot x$), etc
    - Up to 4 versions of each (S/D/C/Z), 46 routines, 3300 LOC
  - Goals
    - Common "pattern" to ease programming, readability
    - Robustness, via careful coding (avoiding over/underflow)
    - Portability + Efficiency via machine specific implementations
  - Why BLAS 1 ? They do $O(n^1)$ ops on $O(n^1)$ data
  - Used in libraries like LINPACK (for linear systems)
    - Source of the name "LINPACK Benchmark" (not the code!)

A brief history of (Dense) Linear Algebra software

- But the BLAS-1 weren't enough
  - Consider AXPY ($y = \alpha \cdot x + y$): 2n flops on 3n read/writes
  - Computational intensity = (2n)/(3n) = 2/3
  - Too low to run near peak speed (read/write dominates)
  - Hard to vectorize ("SIMD'ize") on supercomputers of the day (1980s)

- So the BLAS-2 were invented (1984-1986)
  - Standard library of 25 operations (mostly) on matrix/vector pairs
    - "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot y^T$, $x = T^{-1} \cdot x$
    - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC
  - Why BLAS 2 ? They do $O(n^2)$ ops on $O(n^2)$ data
  - So computational intensity still just $\sim(2n^2)/(n^2) = 2$
    - OK for vector machines, but not for machine with caches

A brief history of (Dense) Linear Algebra software

- The next step: BLAS-3 (1987-1988)
  - Standard library of 9 operations (mostly) on matrix/matrix pairs
    - "GEMM": $C = \alpha \cdot A \cdot B + \beta \cdot C$, $C = \alpha \cdot A \cdot A^T + \beta \cdot C$, $B = T^{-1} \cdot B$
    - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3 ? They do $O(n^3)$ ops on $O(n^2)$ data
  - So computational intensity $(2n^3)/(4n^2) = n/2$ – big at last!
    - Good for machines with caches, other mem. hierarchy levels

- How much BLAS1/2/3 code so far? (all at www.netlib.org/blas)
  - Source: 142 routines, 31K LOC, Testing: 28K LOC
    - Reference (unoptimized) implementation only
  - Part of standard math libraries (e.g. Intel MKL)

A brief history of (Dense) Linear Algebra software

- LAPACK – "Linear Algebra PACKage" - uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1
    - How do we reorganize GE to use BLAS-3 ? (details later)
  - Contents of LAPACK (summary)
    - Algorithms that are (nearly) 100% BLAS 3
      - Linear Systems: solve Ax=b for x
      - Least Squares: choose x to minimize $||Ax-b||_2$
    - Algorithms that are only ≈50% BLAS 3
      - Eigenproblems: Find λ and x where $Ax = \lambda x$
      - Singular Value Decomposition (SVD)
    - Generalized problems (e.g. $Ax = \lambda Bx$)
    - Error bounds for everything
    - Lots of variants depending on A's structure (banded, $A=A^T$, etc)
  - Ongoing development

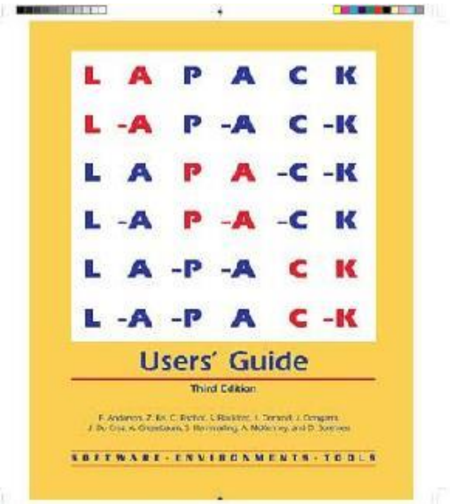
- Is LAPACK parallel?
  - Only if the BLAS are parallel (possible in shared memory)

- ScaLAPACK – "Scalable LAPACK" (1995 – now)
  - For distributed memory – uses MPI
  - More complex data structures, algorithms than LAPACK
  - All at www.netlib.org/scalapack

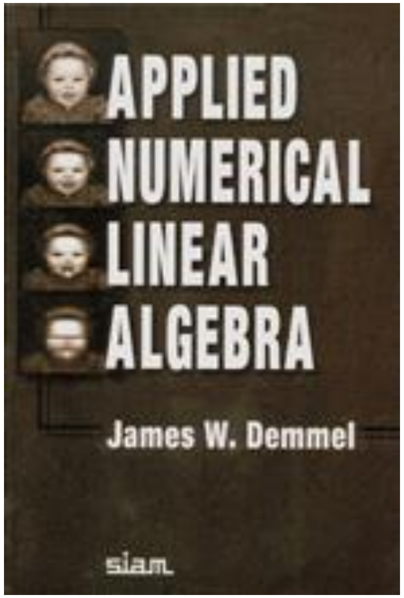
A brief future look at (Dense) Linear Algebra software

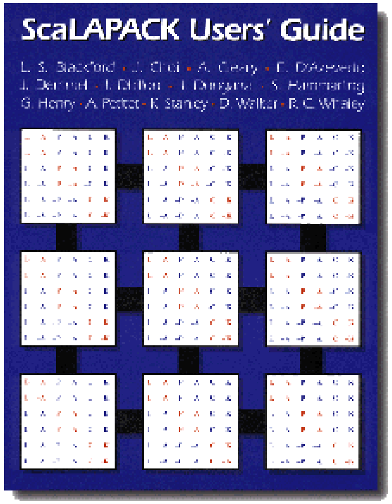
- PLASMA, DPLASMA and MAGMA (now)
  - Ongoing extensions to Multicore/GPU/Heterogeneous
  - Can one software infrastructure accommodate all algorithms and platforms of current (future) interest?
    - How much code generation and tuning can we automate?
  - icl.cs.utk.edu/{{d}plasma,magma}
- Other related projects
  - Elemental (libelemental.org)
    - Distributed memory dense linear algebra
    - "Balance ease of use and high performance"
  - FLAME (z.cs.utexas.edu/wiki/flame.wiki/FrontPage)
    - Formal Linear Algebra Method Environment
    - Attempt to automate code generation across multiple platforms
  - BLAST Forum (www.netlib.org/blas/blast-forum)
    - Attempt to extend BLAS, add new functions, extra-precision, ...

Organizing Linear Algebra – in books

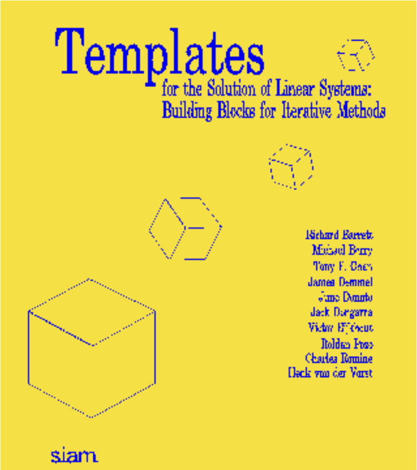


www.netlib.org/lapack

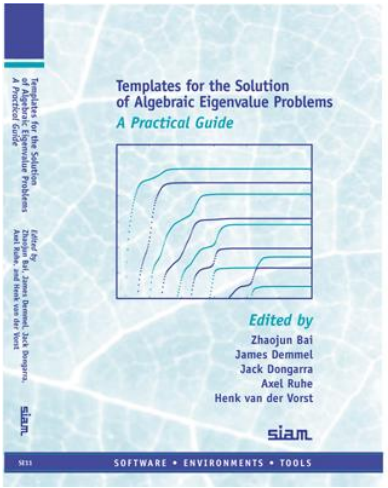




www.netlib.org/scalapack



gams.nist.gov



www.netlib.org/templates

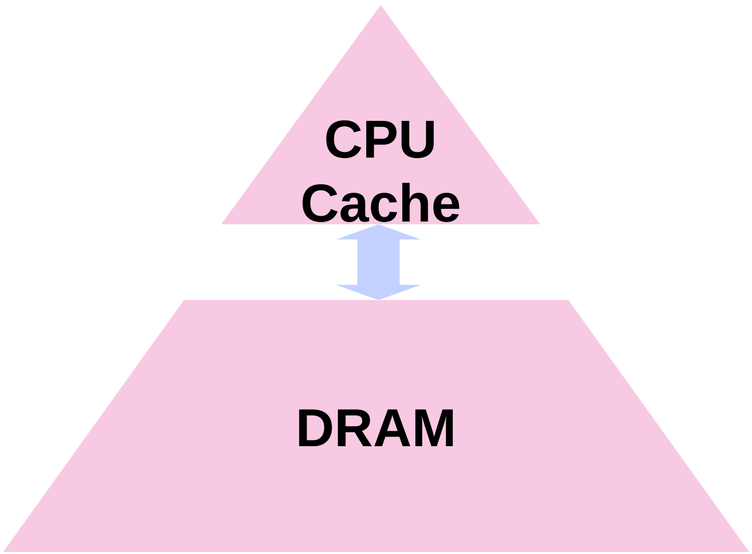
www.cs.utk.edu/~dongarra/etemplates

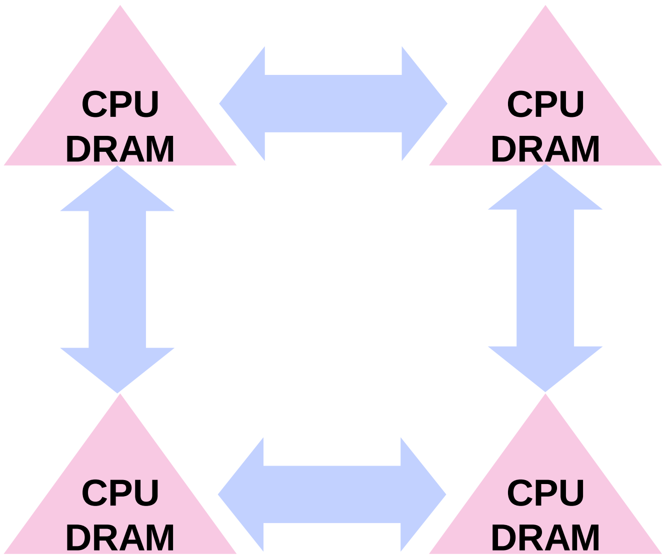
Lower Bounds on Communication

Why avoiding communication is important

Algorithms have two costs:

1. Arithmetic (FLOPS)
2. Communication: moving data between
    - levels of a memory hierarchy (sequential case)
    - processors over a network (parallel case).





Why avoiding communication is important

- Recall $\alpha - \beta - \gamma$ model
- Running time sum of 3 terms:
  - # flops x time per flop
  - # words moved / bandwidth
  - # messages x latency

- Time per flop << 1/bandwidth << latency
  - Gaps growing exponentially in time

Goal: Organize Linear Algebra to Avoid Communication

- Between all memory hierarchy levels
  - L1 $\longleftrightarrow$ L2 $\longleftrightarrow$ DRAM $\longleftrightarrow$ network, etc
- Not just *hiding* communication (overlap with arithmetic)
  - Speedup $\leq$ 2x
- Arbitrary speedups/energy savings possible
- Later: Same goal for other computational patterns
  - Lots of open problems

Review: Blocked Matrix Multiply

- Blocked Matmul $C = A \cdot B$ breaks A, B and C into blocks with dimensions that depend on cache size

… Break $A\ (n \times n)$, $B\ (n \times n)$, $C\ (n \times n)$ into $b \times b$ blocks labeled $A(i,j)$, etc.

… b chosen so 3 $b \times b$ blocks fit in cache

for $i = 1$ *to* n/b, for $j = 1$ *to* n/b, for $k = 1$ *to* n/b

$C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$ … $b \times b$ matmul, $4b^2$ reads/writes

- When $b = 1$, get "naïve" algorithm, want b larger …
- $(n/b)3 \cdot 4b^2 = 4n^3/b$ reads/writes altogether
- Minimized when $3b^2 =$ cache size $= M$, yielding $O(n^3/M^{1/2})$ reads/writes
- What if we had more levels of memory? (L1, L2, cache etc)?
  - Would need 3 more nested loops per level
  - Recursive (cache-oblivious algorithm) also possible

Communication Lower Bounds: Prior Work on Matmul

- Assume n^3 algorithm (i.e., not Strassen-like)
- Sequential case, with fast memory of size M:
  - Lower bound on #words moved to/from slow memory $= \Omega\left(\frac{n^3}{M^{1/2}}\right)$ [Hong, Kung, 81]
  - Attained using blocked or cache-oblivious algorithms

- Parallel case on p processors:
  - Let M be memory per processor; assume load balanced
  - Lower bound on #words moved $= \Omega\left(\frac{n^3}{pM^{1/2}}\right)$ [Irony, Tiskin, Toledo, 04]
  - If $M = 3n^2/p$ (one copy of each matrix), then lower bound $= \Omega\left(\frac{n^2}{p^{1/2}}\right)$
  - Attained by SUMMA, Cannon's algorithm

New lower bound for all "direct" linear algebra

Let M = "fast" memory size per processor
= cache size (sequential case) or $O(n^2/p)$ (parallel case)
#flops = number of flops done per processor

#words_moved per processor = Ω(#flops / $M^{1/2}$)

Lower bound on messages = lower bound on words moved / largest possible message size:

#messages_sent per processor = Ω(#flops / $M^{3/2}$)

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, ...
  - Some whole programs (sequences of these operations, no matter how they are interleaved, e.g., computing A^k)
  - Dense *and* sparse matrices (where #flops $<<$ n^3)
  - Sequential *and* parallel algorithms
  - Some graph-theoretic algorithms (e.g., Floyd-Warshall)
- Generalizations later (Strassen-like algorithms, loops accessing arrays)

New lower bound for all "direct" linear algebra

Let M = "fast" memory size per processor
= cache size (sequential case) or $O(n^2/p)$ (parallel case)
#flops = number of flops done per processor

#words_moved per processor = Ω(#flops / $M^{1/2}$)

Lower bound on messages = lower bound on words moved / largest possible message size:

#messages_sent per processor = Ω(#flops / $M^{3/2}$)

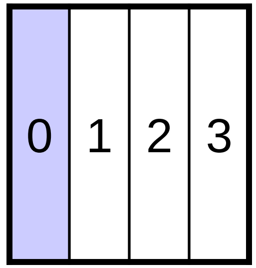
- Sequential case, dense $n \times n$ matrices, so $O(n^3)$ flops
  - #words_moved = $\Omega(n^3/M^{1/2})$
  - #messages_sent = $\Omega(n^3/M^{3/2})$
- Parallel case, dense $n \times n$ matrices
  - Assume load balanced, so $O(n^3/p)$ flops/processor
  - One copy of data, load balanced, so $M = O(n^2/p)$ per processor
  - #words_moved = $\Omega(n^2/p^{1/2})$
  - #messages_sent = $\Omega\left(p^{1/2}\right)$

Can we attain these lower bounds?

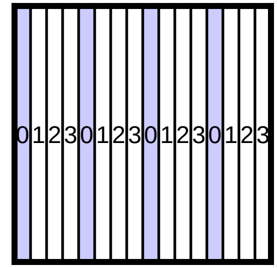
- Do conventional dense algorithms as implemented in LAPACK and ScaLAPACK attain these bounds?
  - Mostly not yet, work in progress
- If not, are there other algorithms that do?
  - Yes
- Goals for algorithms:
  - Minimize #words_moved
  - Minimize #messages_sent
  - Minimize for multiple memory hierarchy levels
  - Fewest flops when matrix fits in fastest memory
- Attainable for nearly all dense linear algebra
  - Just a few prototype implementations so far
  - Only a few sparse algorithms so far (e.g., Cholesky)

Parallel Matrix Multiply

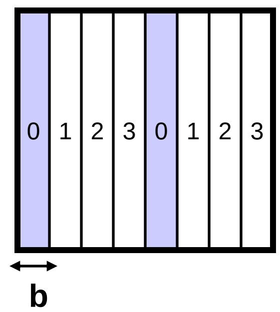
Different Parallel Data Layouts for Matrices (not all!)



1) 1D Column Blocked Layout

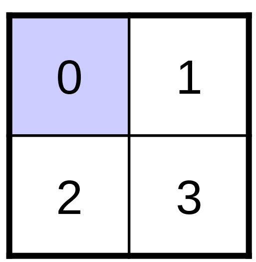


2) 1D Column Cyclic Layout

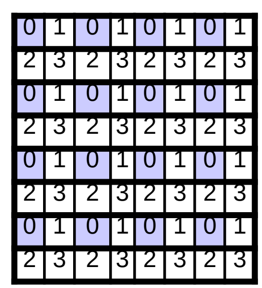


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout

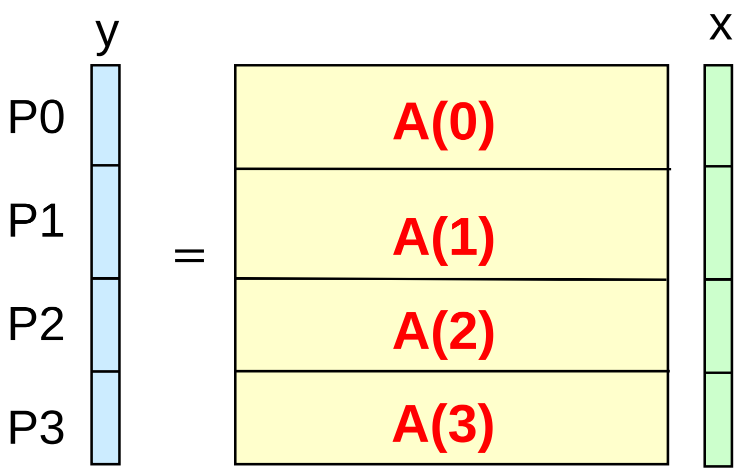


Generalizes others

6) 2D Row and Column Block Cyclic Layout

Parallel Matrix-Vector Product

- Compute $y = y + Ax$, where A is a dense matrix
- Layout:
  - **1D row blocked**
- $A(i)$ refers to the n/p by n block row that processor i owns,
- $x(i)$ and $y(i)$ similarly refer to segments of x, y owned by i
- **Algorithm:**
  - For each processor i
    - Broadcast $x(i)$
    - Compute $y(i) = A(i) \cdot x$



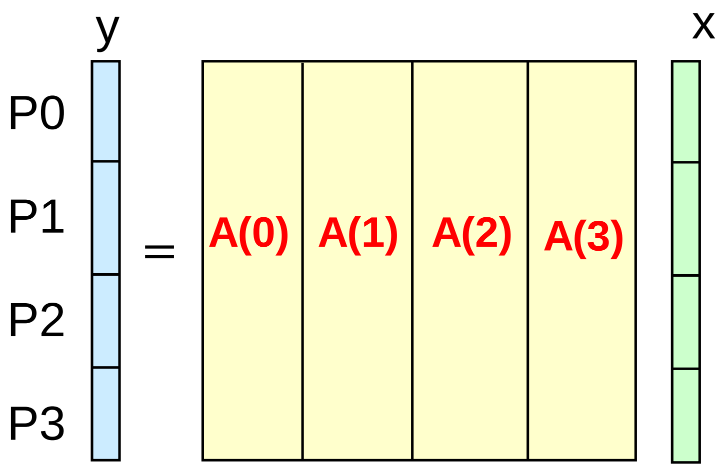
- Algorithm uses the formula

$$y(i) = y(i) + A(i) \cdot x = y(i) + \sum_i A(i,j) \cdot x(j)$$

Parallel Matrix-Vector Product

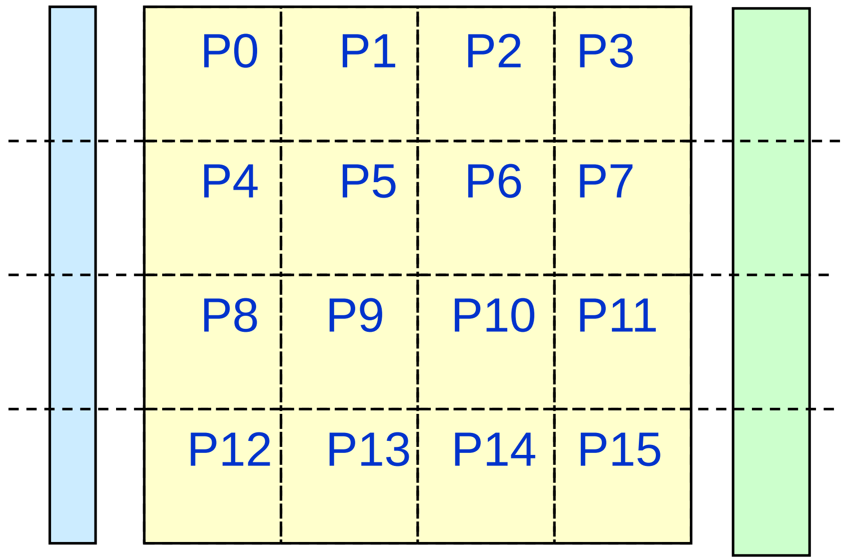
- Compute $y = y + Ax$, where A is a dense matrix
- Layout:
  - **1D column blocked**
- $A(i)$ refers to the n by n/p block column that processor i owns,
- $x(i)$ and $y(i)$ similarly refer to segments of x, y owned by i
- **Algorithm:**
  - For each processor i
    - Compute $y(i) = A(i) \cdot x(i)$
    - Reduction to compute y
- Algorithm uses the formula

$$y = y + \sum_i A(i) \cdot x(i)$$



Matrix-Vector Product $y = y + Ax$

- A 2D blocked layout uses a broadcast and reduction, both on a subset of processors
  - sqrt(p) for square processor grid



Parallel Matrix Multiply

- Computing $C = C + A \cdot B$
- Using basic algorithm: $2n^3$ Flops
- Variables are:
  - Data layout: 1D? 2D? Other?
  - Topology of machine: Ring? Torus?
  - Scheduling communication

- Use of performance models for algorithm design
  - **Message Time = "latency" + #words * time-per-word**

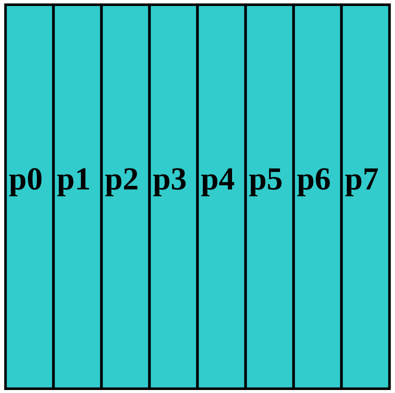
$$= \alpha + n\beta$$

- Efficiency (in any model):
  - serial time / ($p \times$ parallel time)
  - perfect (linear) speedup $\leftrightarrow$ efficiency $= 1$

Matrix Multiply with 1D Column Layout

- Assume matrices are $n \times n$ and n is divisible by p

May be a reasonable assumption for analysis, not for code



- $A(i)$ refers to the n by n/p block column that processor i owns (similiarly for $B(i)$ and $C(i)$)
- $B(j,i)$ is the n/p by n/p subblock of $B(i)$
  - in rows $j \times n/p$ through $(j+1) \times n/p - 1$
- Algorithm uses the formula

$$C(i) = C(i) + A \cdot B(i) = C(i) + \sum_j A(j) \cdot B(j,i)$$

Matrix Multiply: 1D Layout on Bus or Ring

- Algorithm uses the formula

$$C(i) = C(i) + A \cdot B(i) = C(i) + \sum_j A(j) \cdot B(j,i)$$

- First consider a bus-connected machine without broadcast: only one pair of processors can communicate at a time (ethernet)

- Second consider a machine with processors on a ring: all processors may communicate with nearest neighbors simultaneously

MatMul: 1D layout on Bus w/out Broadcast

Naïve algorithm:

$C(myproc) = C(myproc) + A(myproc) \cdot B(myproc, myproc)$

for $i = 0$ to $p - 1$

  for $j = 0$ to $p - 1$ except i

    if $(myproc == i)$ send $A(i)$ to processor j

    if $(myproc == j)$

      receive $A(i)$ from processor i

      $C(myproc) = C(myproc) + A(i) \cdot B(i, myproc)$

    barrier

Cost of inner loop:

computation: $A(i) \cdot B(i, myproc)$: $2n(n/p)2 = 2n^3/p^2$

communication: send $A(i)$: $\alpha + \beta n^2/p$

Naïve MatMul (continued)

Cost of inner loop:

computation: $A(i) \cdot B(i, myproc)$: $2n(n/p)2 \; = \; 2n^3/p^2$

communication: send $A(i)$: $\alpha + \beta n^2/p$

Only 1 pair of processors (i and j) are active on any iteration,
and of those, only i is doing computation

=> the algorithm is almost entirely serial

Running time:

$= p(p-1) \times$ computation $+ \; p(p-1) \times$ communication

$\approx 2n^3 \; + p^2\alpha + pn^2\beta$

This is worse than the serial time and grows with p.

Matmul for 1D layout on a Processor Ring

- Pairs of adjacent processors can communicate simultaneously

```
Copy A(myproc) into Tmp
C(myproc) = C(myproc) + Tmp*B(myproc , myproc)
for j = 1 to p-1
    Send Tmp to processor myproc+1 mod p
    Receive Tmp from processor myproc-1 mod p
    C(myproc) = C(myproc) + Tmp*B( myproc-j mod p , myproc)
```

- Time of inner loop = $2(\alpha + \beta n^2/p) + 2n(n/p)^2$

Matmul for 1D layout on a Processor Ring

- Time of inner loop = $2(\alpha + \beta n^2/p) + 2n(n/p)2$
- Total Time $= 2n(n/p)^2 + (p-1) \times$ Time of inner loop

  $\approx 2n^3/p + 2p\alpha + 2\beta n^2$

- (Nearly) Optimal for 1D layout on Ring or Bus, even with Broadcast:
  - Perfect speedup for arithmetic
  - A(myproc) must move to each other processor, costs at least

    $(p-1) \times$ (cost of sending $n \times (n/p)$ words)

- Parallel Efficiency = $2n^3/(p \times \text{Total Time})$

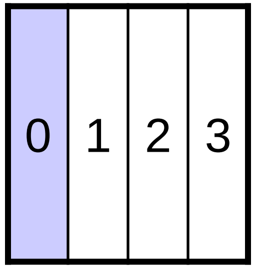
  $= 1/(1 + \alpha p^2/(2n^3) + \beta p/(2n))$

  $= 1/(1 + O(p/n))$

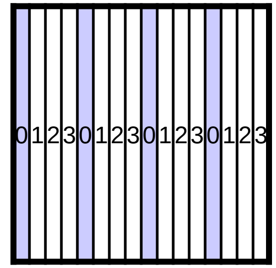
- Grows to 1 as n/p increases (or α and β shrink)

- But far from communication lower bound

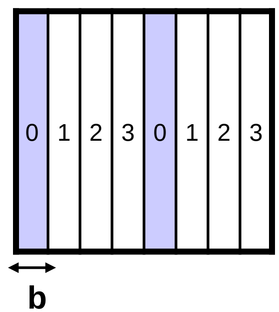
Need to try 2D Matrix layout



1) 1D Column Blocked Layout

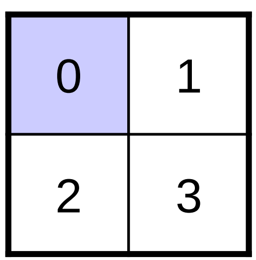


2) 1D Column Cyclic Layout

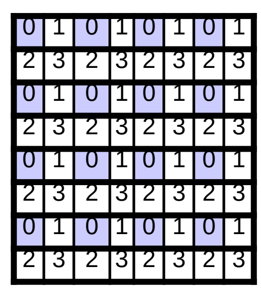


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout



Generalizes others

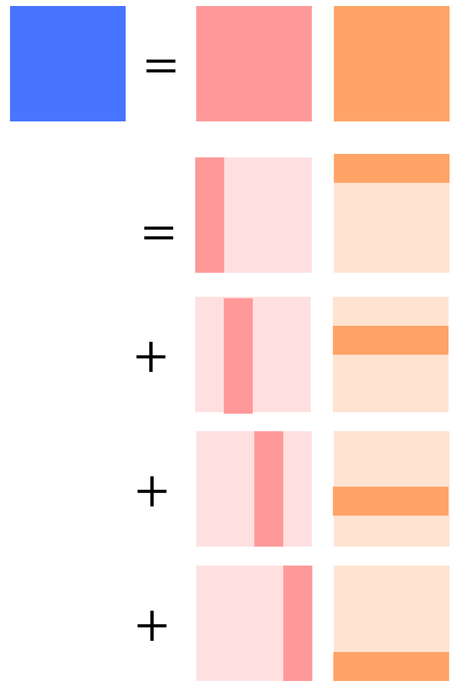
6) 2D Row and Column Block Cyclic Layout

Summary of Parallel Matrix Multiply

- SUMMA
  - Scalable Universal Matrix Multiply Algorithm
  - Attains communication lower bounds (within $\log p$)
- Cannon
  - Historically first, attains lower bounds
  - More assumptions
    - A and B square
    - p a perfect square
- 2.5D SUMMA
  - Uses more memory to communicate even less
- Parallel Strassen
  - Attains different, even lower bounds

SUMMA uses Outer Product form of MatMul

- $C = A \cdot B$ means $C(i,j) = \sum_k A(i,k) \cdot B(k,j)$



- Column-wise outer product:

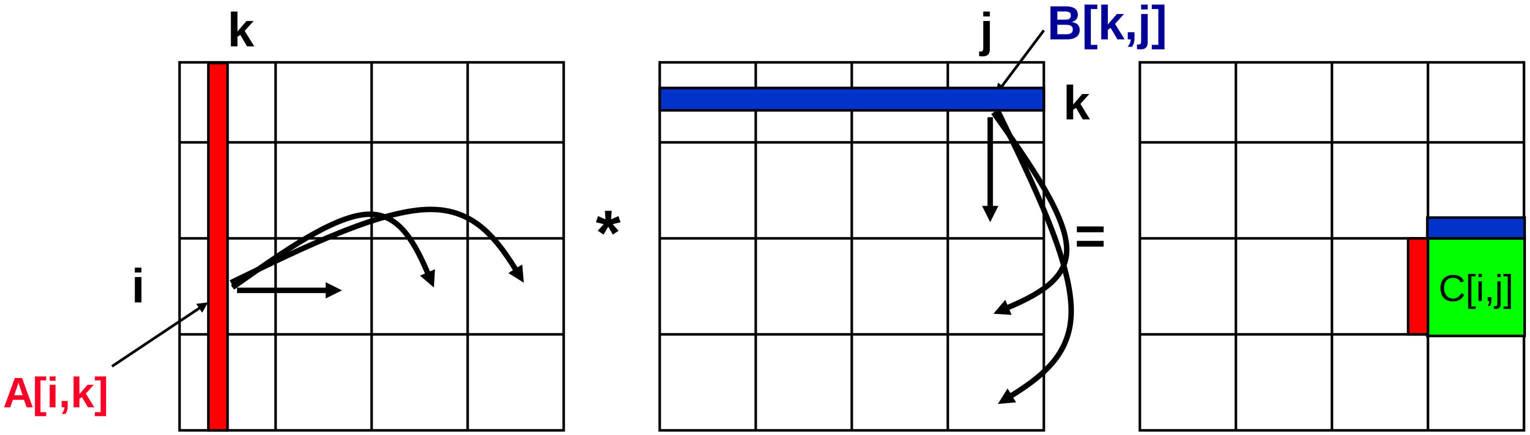
$$C = A \cdot B$$
$$= \sum_k A(:,k) \cdot B(k,:)$$
$$= \sum_k \left(k^{th}\ col\ of\ A\right) \cdot \left(k^{th}\ row\ of\ B\right)$$

- Block column-wise outer product

  (block size = 4 for illustration)

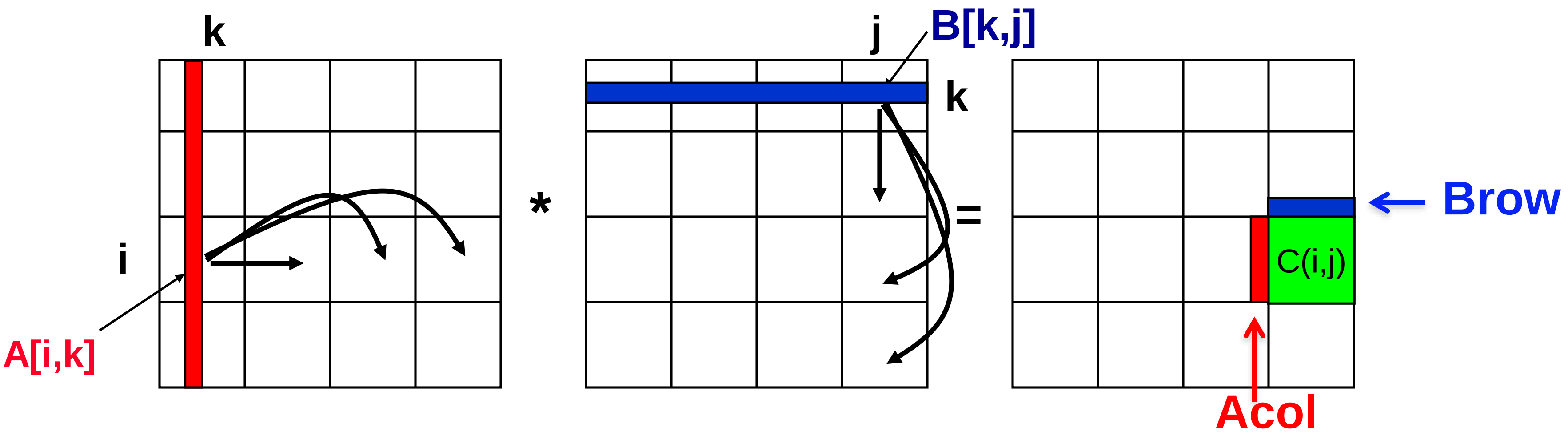
$$C = A \cdot B$$
$$= A(:,1{:}4) \cdot B(1{:}4,:) + A(:,5{:}8) \cdot B(5{:}8,:) + \ldots$$
$$= \sum_k \left(k^{th}\ block\ of\ 4\ cols\ of\ A\right) \cdot \left(k^{th}\ block\ of\ 4\ rows\ of\ B\right)$$

SUMMA – $n \times n$ matmul on $p^{1/2} \times p^{1/2}$ grid



- $C[i,j]$ is $n/p^{1/2} \times n/p^{1/2}$ submatrix of C on processor p_{ij}
- $A[i,k]$ is $n/p^{1/2} \times b$ submatrix of A
- $B[k,j]$ is $b \times n/p^{1/2}$ submatrix of B
- $C[i,j] = C[i,j] + \sum_k A[i,k] \cdot B[k,j]$
- summation over submatrices
- Need not be square processor grid

SUMMA – $n \times n$ matmul on $p^{1/2} \times p^{1/2}$ grid



For $k = 0$ to $n/b - 1$

for all $i = 1$ to $p^{1/2}$

owner of A[i,k] broadcasts it to whole processor row (using binary tree)

for all $j = 1$ to $p^{1/2}$

owner of B[k,j] broadcasts it to whole processor column (using binary tree)

Receive A[i,k] into Acol

Receive B[k,j] into Brow

C_myproc = C_myproc + Acol * Brow

SUMMA Costs

For $k = 0$ to $n/b - 1$

for all $i = 1$ to $p^{1/2}$

owner of A[i,k] broadcasts it to whole processor row (using binary tree)

… #words $= \log p^{1/2} \times b \times n/p^{1/2}$, #messages $= \log p^{1/2}$

for all $j = 1$ to $p^{1/2}$

owner of B[k,j] broadcasts it to whole processor column (using binary tree)

… same #words and #messages

Receive A[i,k] into Acol

Receive B[k,j] into Brow

C_myproc = C_myproc + Acol * Brow … #flops = $2n^2 * b/p$

- Total #words $= \log p \times n^2/p^{1/2}$
  - Within factor of $\log p$ of lower bound
  - (more complicated implementation removes $\log p$ factor)
- Total #messages $= \log p \times n/b$
  - Choose b close to maximum, $n/p^{1/2}$, to approach lower bound $p^{1/2}$
  - Total #flops $= 2n^3/p$

Can we do better?

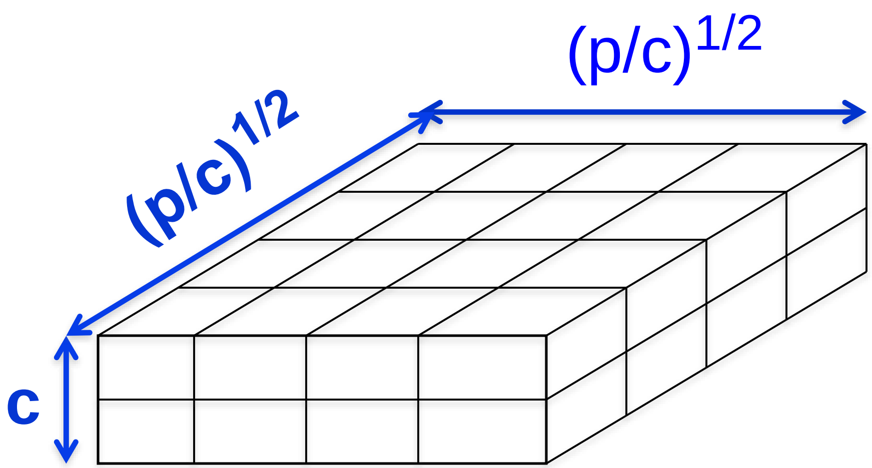
- Lower bound assumed 1 copy of data: $M = O(n^2/p)$ per proc.
- What if matrix small enough to fit $c > 1$ copies, so $M = cn^2/p$?
  - #words_moved $= \Omega(\#flops/M^{1/2}) = \Omega(n^2/c^{1/2}p^{1/2})$
  - #messages $= \Omega(\#flops/M^{3/2}) = \Omega(p^{1/2}/c^{3/2})$

- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = p^{1/3}$
    - Bernsten 89, Agarwal, Chandra, Snir 90, Aggarwal 95
    - Processors arranged in $p^{1/3} \times p^{1/3} \times p^{1/3}$ grid
    - Processor (i,j,k) performs $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$, where each submatrix is $n/p^{1/3} \times n/p^{1/3}$

  - Not always that much memory available…

2.5D Matrix Multiplication

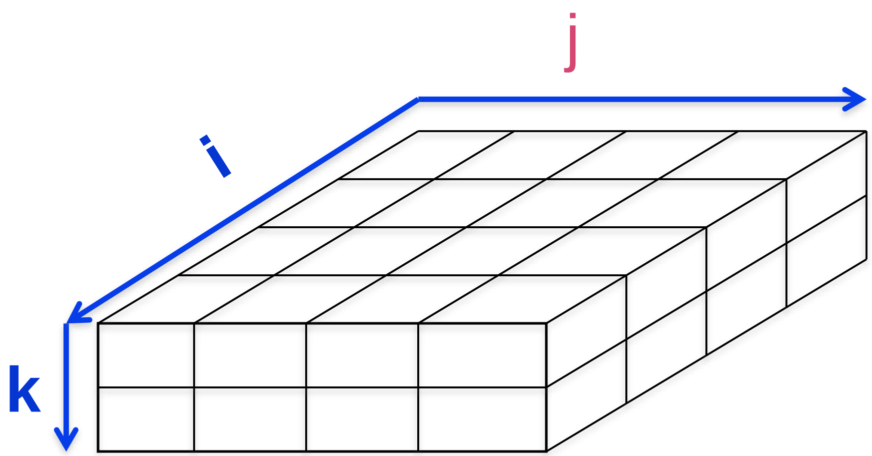
- Assume can fit cn^2/p data per processor, $c > 1$
- Processors form $(p/c)^{1/2} \times (p/c)^{1/2} \times c$ grid



Example: $p = 32,\ c = 2$

2.5D Matrix Multiplication

- Assume can fit cn^2/p data per processor, $c > 1$
- Processors form $(p/c)^{1/2} \times (p/c)^{1/2} \times c$ grid

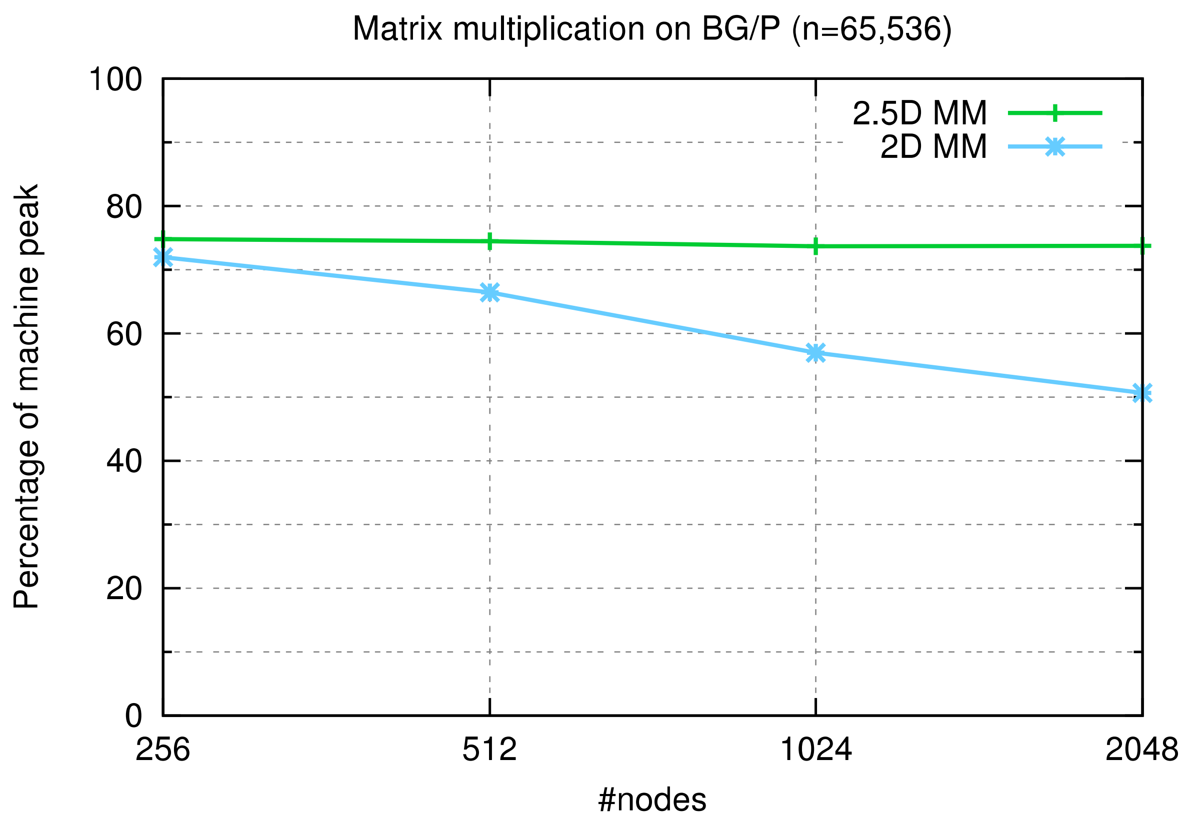


Initially p(i,j,0) owns A(i,j) and B(i,j) each of size $n(c/p)^{1/2} \times n(c/p)^{1/2}$

(1) p(i,j,0) broadcasts A(i,j) and B(i,j) to p(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so p(i,j,0) owns C(i,j)

2.5D Matmul on IBM BG/P, n=64K

- As p increases, available memory grows → c increases proportionally to p
  - #flops, #words_moved, #messages per proc all decrease proportionally to p
  - #words_moved $= \Omega(\#flops/M^{1/2}) = \Omega(n^2/(c^{1/2}p^{1/2}))$
  - #messages $= \Omega(\#flops/M^{3/2}) = \Omega(p^{1/2}/c^{3/2})$
- Perfect strong scaling! But only up to $c = p^{1/3}$



Classical Matmul *vs*

- Complexity of classical Matmul
- Flops: $O(n^3/p)$
- Communication lower bound on #words:

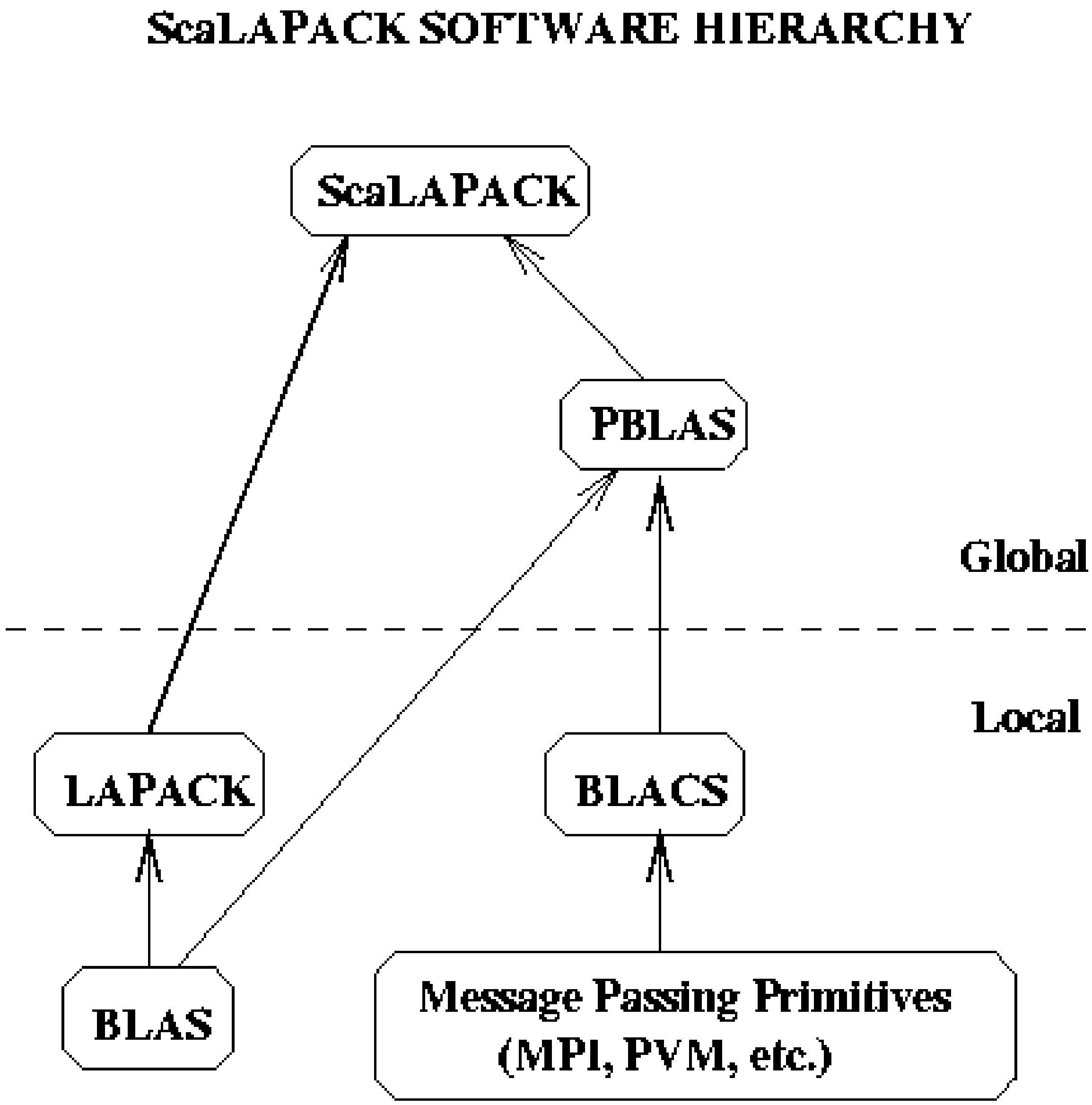
  $\Omega((n^3/p)/M^{1/2}) = \Omega(M(n/M^{1/2})^3/p)$
- Communication lower bound on #messages:

  $\Omega((n^3/p)/M^{3/2}) = \Omega((n/M^{1/2})^3/p$
- All attainable as M increases past $O(n^2/p)$, up to a limit:

  can increase M by factor up to $p^{1/3}$

  #words as low as $\Omega(n/p^{2/3})$

ScaLAPACK Parallel Library



Extensions of Lower Bound and Optimal Algorithms

- For each processor that does G flops with fast memory of size M

  $$\#\text{words\_moved} = \Omega(G/M^{1/2})$$

- Extension: for any program that looks like
  - Nested loops …
  - That access arrays …
  - Where array subscripts are linear functions of loop indices
    - Ex: A(i,j), B(3*i-4*k+5*j, i-j, 2*k, …), …
  - There is a constant s such that

  $$\#\text{words\_moved} = \Omega(G/M^{s-1})$$

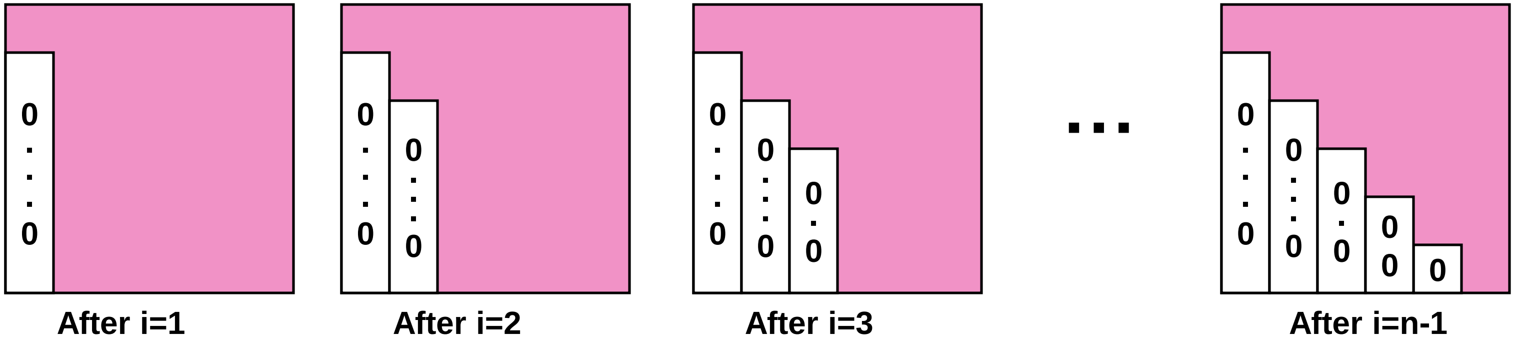
  - s comes from recent generalization of Loomis-Whitney ($s = 3/2$)
  - Ex: linear algebra, n-body, database join, …
  - Lots of open questions: deriving s, optimal algorithms …

LU and QR Factorizations

Gaussian Elimination (GE) for solving Ax=b

- Add multiples of each row to later rows to make A upper triangular
- Solve resulting triangular system Ux = c by substitution

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
    … for each row j below row i
    for j = i+1 to n
        … add a multiple of row i to row j
        tmp = A(j,i);
        for k = i to n
            A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```



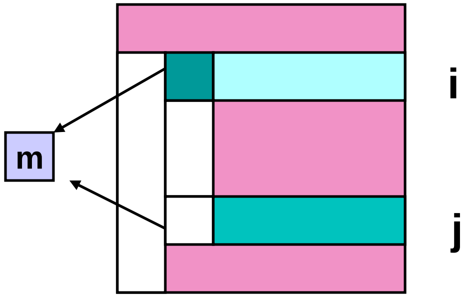
Refine GE Algorithm (1/5)

- Initial Version

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
   … for each row j below row i
   for j = i+1 to n
      … add a multiple of row i to row j
      tmp = A(j,i);
      for k = i to n
          A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

- Remove computation of constant tmp/A(i,i) from inner loop.

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```



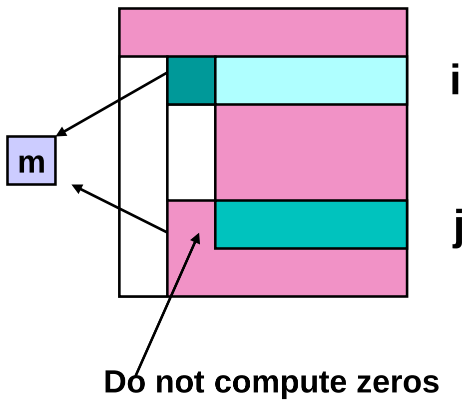
Refine GE Algorithm (2/5)

- Last version

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```

- Don't compute what we already know: zeros below diagonal in column i

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - m * A(i,k)
```



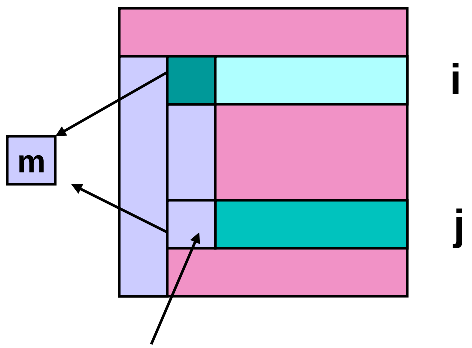
Refine GE Algorithm (3/5)

- Last version

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - m * A(i,k)
```

- Store multipliers m below diagonal in zeroed entries for later use

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



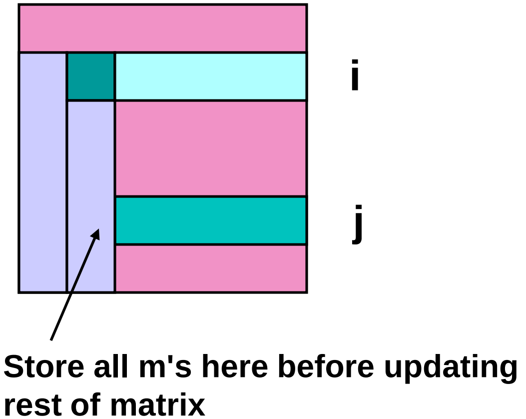
Refine GE Algorithm (4/5)

- Last version

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- Split Loop

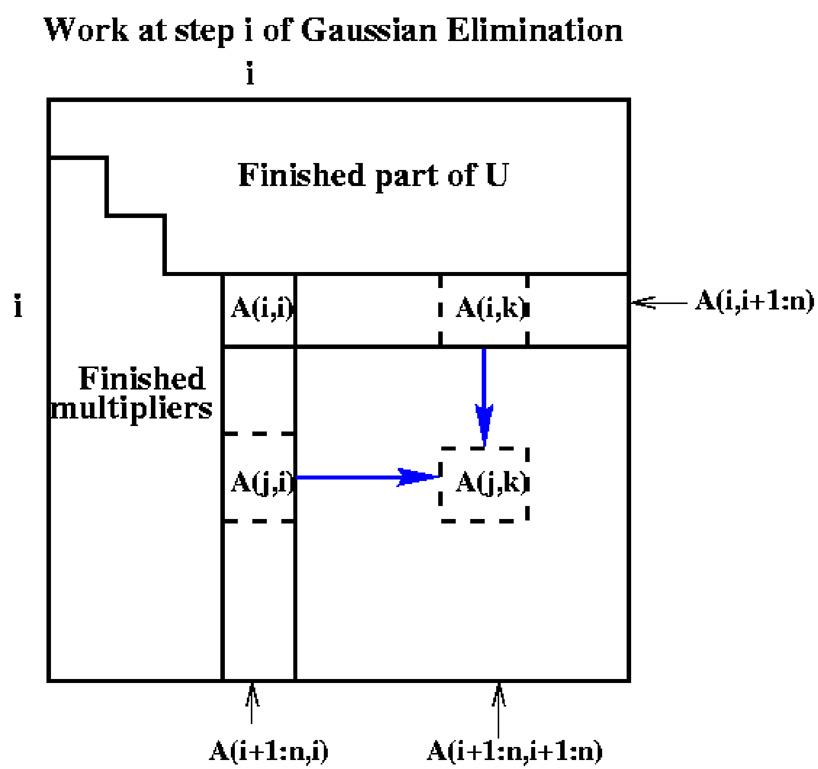
```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
    for j = i+1 to n
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```



Refine GE Algorithm (5/5)

- Last version
- Express using matrix operations (BLAS)

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
    for j = i+1 to n
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

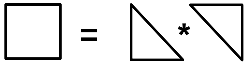



```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) * ( 1 / A(i,i) )
        … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )
        - A(i+1:n , i) * A(i , i+1:n)
        … BLAS 2 (rank-1 update)
```

What GE really computes

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)     … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)   …  BLAS 2 (rank-1 update)
```

- Call the strictly lower triangular matrix of multipliers M, and let L = I+M
- Call the upper triangle of the final matrix U
- *Lemma (LU Factorization):* If the above algorithm terminates (does not divide by zero) then A = LU
- Solving Ax=b using GE

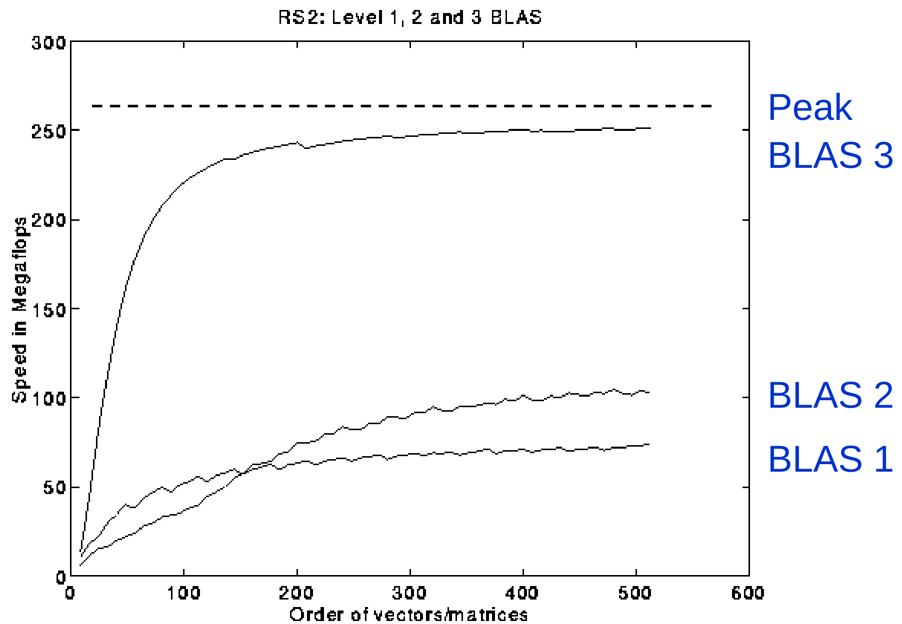


    - Factorize A = LU using GE (cost = 2/3 n^3 flops)
    - Solve Ly = b for y, using substitution (cost = n^2 flops)
    - Solve Ux = y for x, using substitution (cost = n^2 flops)
- Thus Ax = (LU)x = L(Ux) = Ly = b as desired

Problems with basic GE algorithm

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)         … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
              - A(i+1:n , i) * A(i , i+1:n)
```

- What if some A(i,i) is zero? Or very small?
  - Result may not exist, or be "unstable", so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lecture…)



Pivoting in Gaussian Elimination

- $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ fails completely because can't divide by $A(1,1) = 0$

- But solving $Ax = b$ should be easy!

- When diagonal $A(i,i)$ is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - Numerical instability
  - Roundoff error is cause

- Cure: Pivot (swap rows of A) so $A(i,i)$ large

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that A(i,i) is largest in column

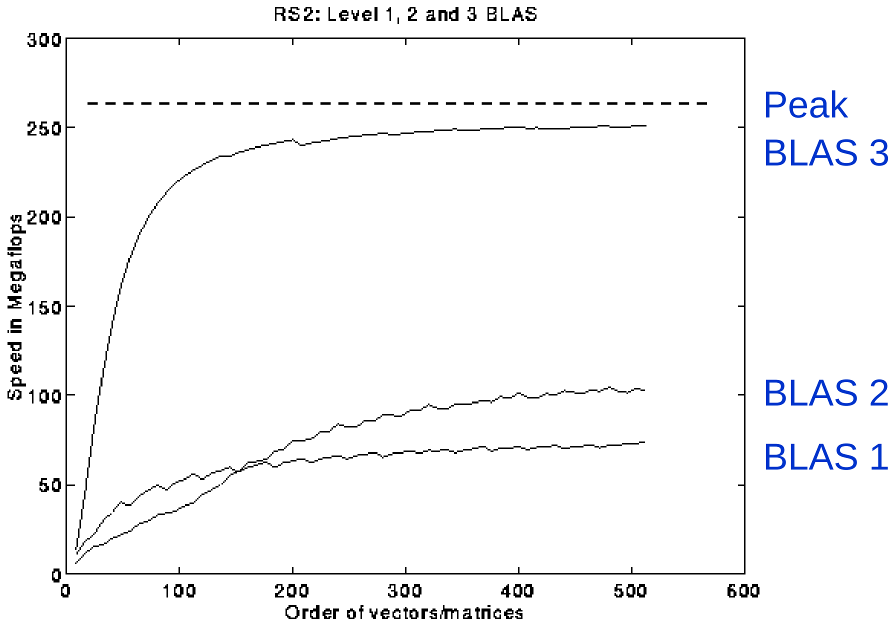
```
for i = 1 to n-1
    find and record k where |A(k,i)| = max{i ≤ j ≤ n} |A(j,i)|
        … i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif  k ≠ i
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)        … each |quotient| ≤ 1
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- *Lemma*: This algorithm computes A = PLU, where P is a permutation matrix.
- This algorithm is numerically stable in practice
- For details see LAPACK code at
  http://www.netlib.org/lapack/single/sgetf2.f
- Standard approach – but communication costs?

Problems with basic GE algorithm

- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
          - A(i+1:n , i) * A(i , i+1:n)
```

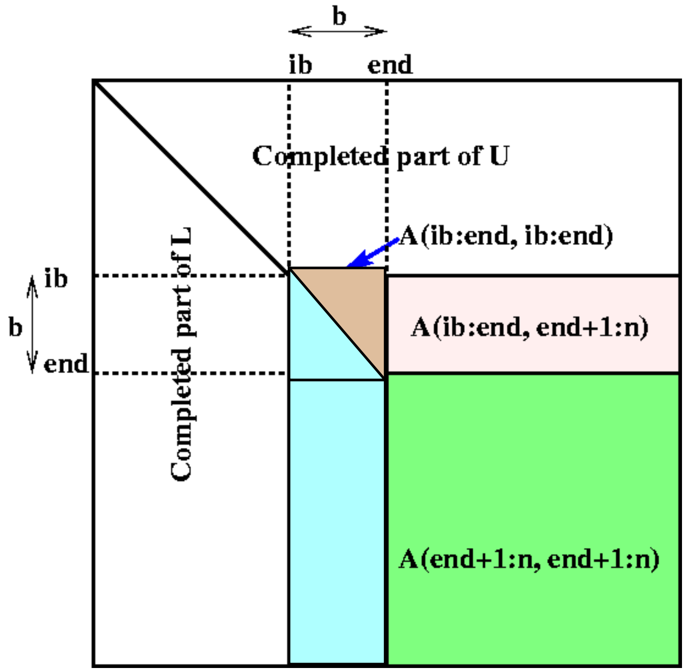


Converting BLAS2 to BLAS3 in GEPP

- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- BIG IDEA: Delayed Updates
  - Save updates to "trailing matrix" from several consecutive BLAS2 (rank-1) updates
  - Apply many updates simultaneously in one BLAS3 (matmul) operation
- Same idea works for much of dense linear algebra
  - Not eigenvalue problems or SVD – need more ideas
- First Approach: Need to choose a block size b
  - Algorithm will save and apply b updates
  - b should be small enough so that active submatrix consisting of b columns of A fits in cache
  - b should be large enough to make BLAS3 (matmul) fast

Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

```
for   ib = 1 to n-1 step b     … Process matrix b columns at a time
    end = ib + b-1             … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                      … apply delayed updates with single matrix-multiply
                      … with inner dimension b
```



Communication Lower Bound for GE

- Matrix multiply can be "reduced to" GE
- Not a good way to do matmul but it shows that GE needs at least as much communication as matmul
- Does blocked GEPP minimize communication?

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

Does LAPACK's GEPP Minimize Communication?

```
for   ib = 1 to n-1 step b     … Process matrix b columns at a time
    end = ib + b-1             … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
                             - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                          … apply delayed updates with single matrix-multiply
                          … with inner dimension b
```

- Case 1: $n \geq M$ - huge matrix – attains lower bound
  - $b = M^{1/2}$ optimal, dominated by matmul
- Case 2: $n \leq M^{1/2}$ - small matrix – attains lower bound
  - Whole matrix fits in fast memory, any algorithm attains lower bound
- Case 3: $M^{1/2} < n < M$ - medium size matrix – not optimal
  - Can't choose b to simultaneously optimize matmul and BLAS2 GEPP of n x b submatrix
  - Worst case: Exceed lower bound by factor $M^{1/6}$ when $n = M^{2/3}$

Explicitly Parallelizing Gaussian Elimination

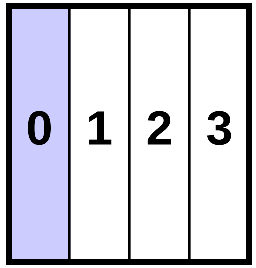
- Parallelization steps
  - Decomposition: identify enough parallel work, but not too much
  - Assignment: load balance work among threads
  - Orchestrate: communication and synchronization
  - Mapping: which processors execute which threads (locality)
- Decomposition
  - In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with n^2 processors, need 3n parallel steps, O(n log n) with pivoting

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
                     - A(i+1:n , i) * A(i , i+1:n)
```

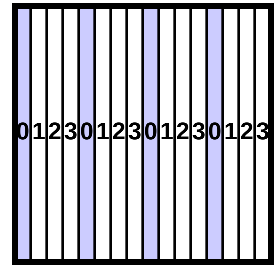
  - This is too fine-grained, prefer calls to local matmuls instead
  - Need to use parallel matrix multiplication
- Assignment and Mapping
  - Which processors are responsible for which submatrices?

Different Data Layouts for Parallel GE

Bad load balance:
P0 idle after first
n/4 steps



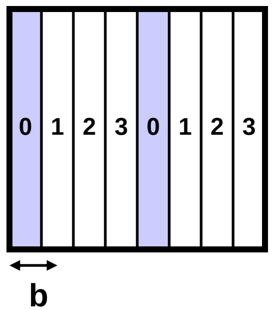
1) 1D Column Blocked Layout



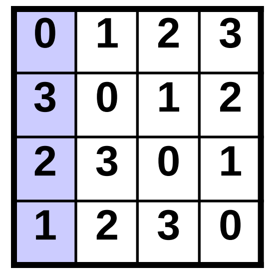
Load balanced, but can't
easily use BLAS3

2) 1D Column Cyclic Layout

Can trade load balance
and BLAS3
performance by
choosing b, but
factorization of block
column is a bottleneck



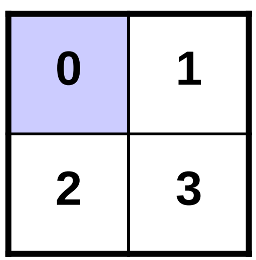
3) 1D Column Block Cyclic Layout



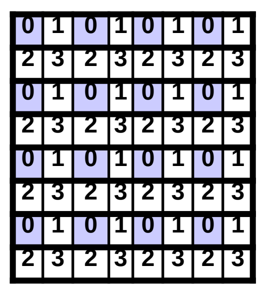
Complicated addressing,
May not want full parallelism
In each column, row

4) Block Skewed Layout

Bad load balance:
P0 idle after first
n/2 steps



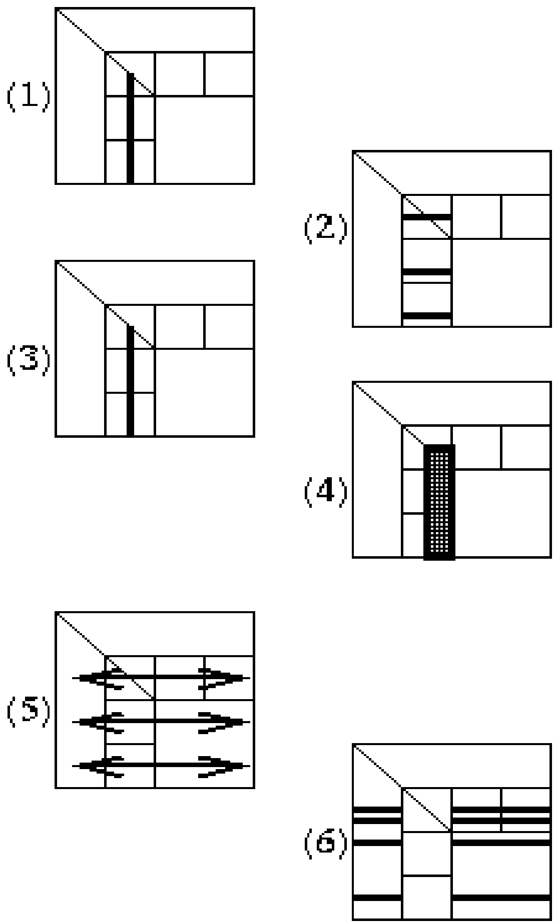
5) 2D Row and Column Blocked Layout



The winner!

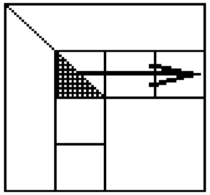
6) 2D Row and Column
Block Cyclic Layout

Distributed GE with a 2D Block Cyclic Layout

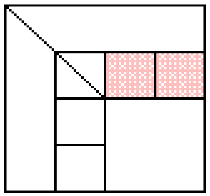



```
for ib = 1 to n-1 step b

    end = min( ib+b-1, n )

    for i = ib to end

    (1)    find pivot row k, column broadcast

    (2)    swap rows k and i in block column, broadcast row k

    (3)    A( i+1:n , i ) = A( i+1:n , i ) / A( i , i )

    (4)    A( i+1:n , i+1:end) -= A( i+1:n , i ) * A( i , i+1:end )

    end for

(5)    broadcast all swap information right and left

(6)    apply all rows swaps to other columns
```

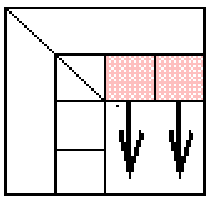
Distributed GE with a 2D Block Cyclic Layout

(7) 

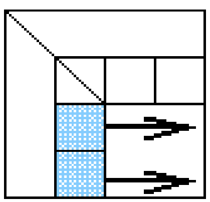
(7) Broadcast LL right

(8) 

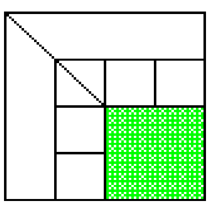
(8) A(ib:end , end+1:n) = LL \ A(ib:end , end+1:n)

(9) 

(9) Broadcast A(ib:end , end+1:n) down

(10) 

(10) Broadcast A(end+1:n , ib:end) right

(11) 

(11) Eliminate A(end+1:n , end+1:n)

Matrix multiply of
green = green - blue * pink

Does ScaLAPACK Minimize Communication?

- Lower Bound: $O(n^2/p^{1/2})$ words sent in $O(p^{1/2})$ mess.
  - Attained by Cannon and SUMMA (nearly) for matmul
- ScaLAPACK:
  - $O(n^2 \log p \,/p^{1/2})$ words sent – close enough
  - $O(n \log p)$ messages – too large
  - Why so many? One reduction costs $O(\log p)$ per column to find maximum pivot, times $n =$ #columns
- Need to replace partial pivoting to reduce #messages
  - Suppose we have $n \times n$ matrix on $p^{1/2} \times p^{1/2}$ processor grid
  - Goal: For each panel of b columns spread over $p^{1/2}$ procs, identify b "good" pivot rows in one reduction
    - Call this factorization TSLU = "Tall Skinny LU"
  - Several natural bad (numerically unstable) ways explored, but good way exists
    - "Communication Avoiding GE", [Demmel, Grigori, Xiang, 2008]

Choosing Rows by "Tournament Pivoting"

$$W^{nxb} = \begin{pmatrix} W_1 \\ \hline W_2 \\ \hline W_3 \\ \hline W_4 \end{pmatrix} = \begin{pmatrix} P_1 \cdot L_1 \cdot U_1 \\ \hline P_2 \cdot L_2 \cdot U_2 \\ \hline P_3 \cdot L_3 \cdot U_3 \\ \hline P_4 \cdot L_4 \cdot U_4 \end{pmatrix}$$

Choose b pivot rows of W_1, call them W_1'
Choose b pivot rows of W_2, call them W_2'
Choose b pivot rows of W_3, call them W_3'
Choose b pivot rows of W_4, call them W_4'

$$\begin{pmatrix} W_1' \\ W_2' \\ \hline W_3' \\ W_4' \end{pmatrix} = \begin{pmatrix} P_{12} \cdot L_{12} \cdot U_{12} \\ \hline P_{34} \cdot L_{34} \cdot U_{34} \end{pmatrix}$$

Choose b pivot rows, call them W_{12}'
Choose b pivot rows, call them W_{34}'

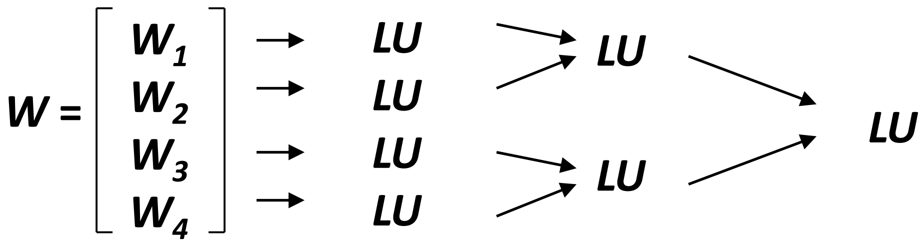
$$\begin{pmatrix} W_{12}' \\ W_{34}' \end{pmatrix} = P_{1234} \cdot L_{1234} \cdot U_{1234}$$

Choose b pivot rows

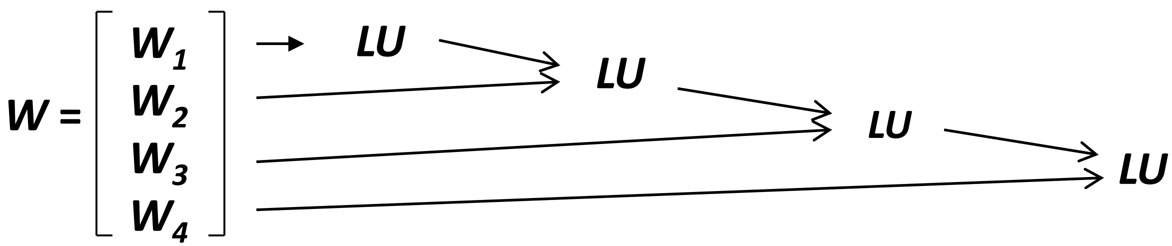
Go back to W and use these b pivot rows
(move them to top, do LU without pivoting)
Not the same pivots rows chosen as for GEPP
Proof that this is numerically stable [Demmel, Grigori, Xiang, '11]

Minimizing Communication in TSLU

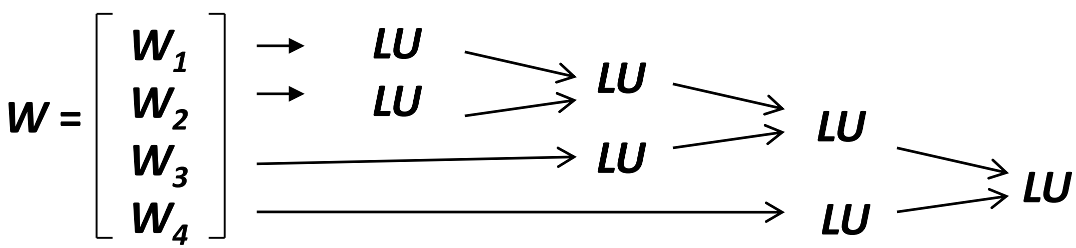
Parallel:



Sequential:



Dual Core:



Multicore / Multisocket / Multirack / Multisite / Out-of-core:?

Can choose reduction tree dynamically

Performance vs. ScaLAPACK LU

- TSLU
  - 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)
- CALU
  - 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)
- See INRIA Tech Report 6523 (2008)

Same idea for TSQR: QR of a Tall, Skinny matrix

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

$$\begin{pmatrix} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01}\, R_{01} \\ \hline Q_{11}\, R_{11} \end{pmatrix}$$

$$\begin{pmatrix} R_{01} \\ \hline R_{11} \end{pmatrix} = \begin{pmatrix} Q_{02}\, R_{02} \end{pmatrix}$$

Same idea for TSQR: QR of a Tall, Skinny matrix

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

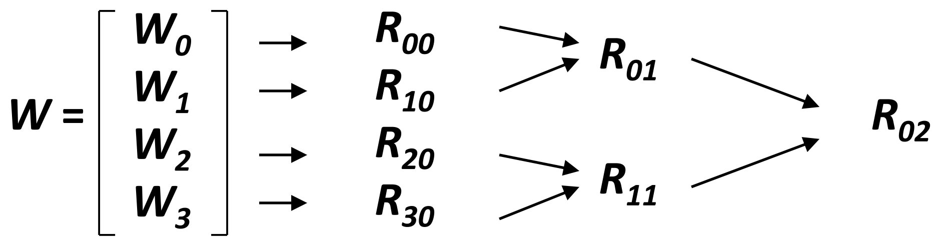
$$\begin{pmatrix} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01} R_{01} \\ \hline Q_{11} R_{11} \end{pmatrix} = \begin{pmatrix} Q_{01} & \\ \hline & Q_{11} \end{pmatrix} \cdot \begin{pmatrix} R_{01} \\ \hline R_{11} \end{pmatrix}$$

$$\begin{pmatrix} R_{01} \\ \hline R_{11} \end{pmatrix} = \begin{pmatrix} Q_{02} R_{02} \end{pmatrix}$$

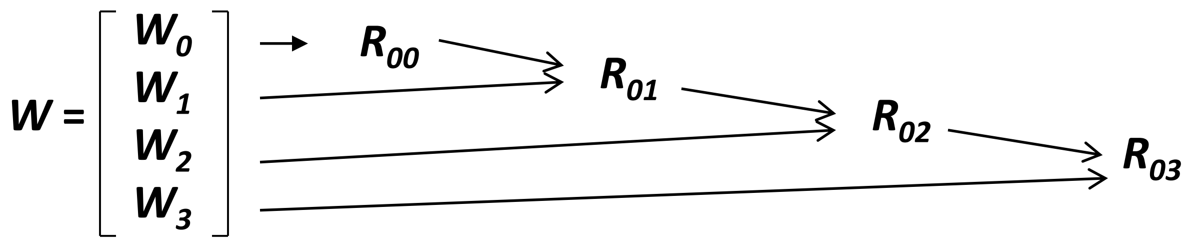
Output = { Q_{00}, Q_{10}, Q_{20}, Q_{30}, Q_{01}, Q_{11}, Q_{02}, R_{02} }

TSQR: An Architecture-Dependent Algorithm

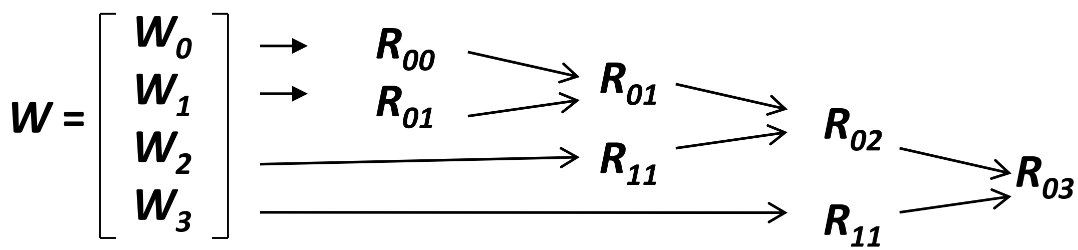
Parallel:



Sequential:



Dual Core:



Multicore / Multisocket / Multirack / Multisite / Out-of-core:?

Can choose reduction tree dynamically

Summary of dense *parallel* $O(n^3/p)$ algorithms attaining comm. lower bounds

- References are from **Table 3.2 in [Ballard, C., Demmel, Hoemmen, Knight, Schwartz, Acta Numerica vol 23, 2014]** (Table 3.1 for sequential algorithms)
- Assume $n \times n$ matrices on p procs, minimum memory per proc: $M = O(n^2/p)$
  - $\#\text{words moved} = \Omega(n^2/p^{1/2})$, $\#\text{messages} = \Omega(p^{1/2})$,
- ScaLAPACK in red
  - ScaLAPACK sends $> n/p^{1/2}$ times too many messages (except Cholesky)

Computation	Minimizes # Words	Minimizes # Messages
BLAS3	[1,**2**,3,4]	[1,**2**,3,4]
Cholesky	[**2**]	[**2**]
LU	[**2**,5,10,11]	[5,10,11]
Symmetric Indefinite	[**2**,6,9]	[6,9]
QR	[**2**,7]	[7]
Eig($A=A^T$) and SVD	[**2**,*8*,9]	[*8*,9]
Eig(A)	[*8*]	[*8*]