CS 267
Dense Linear Algebra: History and Structure, Parallel Matrix Multiplication

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr16

Quick review of earlier lecture

- What do you call
  - A program written in PyGAS, a Global Address Space language based on Python…
  - That uses a Monte Carlo simulation algorithm to approximate π …
  - That has a race condition, so that it gives you a different funny answer every time you run it?

Monte - π - thon

Outline

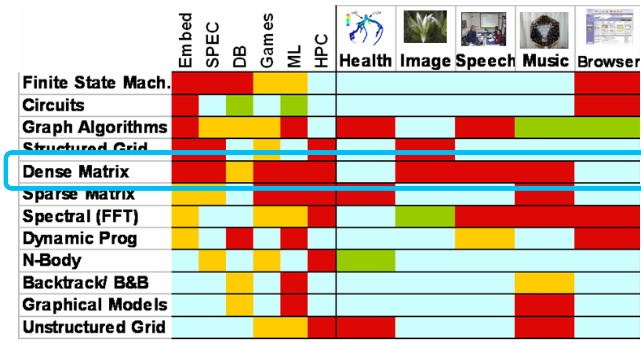
- History and motivation
  - What is dense linear algebra?
  - Why minimize communication?
  - Lower bound on communication
- Parallel Matrix-matrix multiplication
  - Attaining the lower bound
- Other Parallel Algorithms (next lecture)

Outline

- History and motivation
  - What is dense linear algebra?
  - Why minimize communication?
  - Lower bound on communication
- Parallel Matrix-matrix multiplication
  - Attaining the lower bound
- Other Parallel Algorithms (next lecture)

Motifs

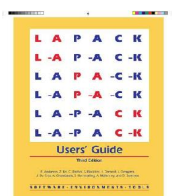
The Motifs (formerly “Dwarfs”) from
“The Berkeley View” (Asanovic et al.)
Motifs form key computational patterns



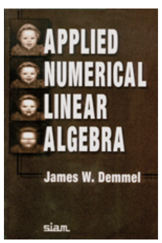
What is dense linear algebra?

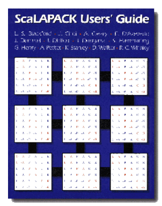
- Not just matmul!
- Linear Systems: Ax=b
- Least Squares: choose x to minimize $||Ax-b||_2$
  - Overdetermined or underdetermined; Unconstrained, constrained, or weighted
- Eigenvalues and vectors of Symmetric Matrices
  - Standard (Ax = λx), Generalized (Ax=λBx)
- Eigenvalues and vectors of Unsymmetric 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 …
  - 27 types in LAPACK (and growing…)
- 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…)

Organizing Linear Algebra – in books

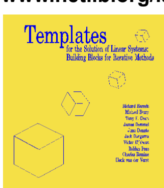


www.netlib.org/lapack

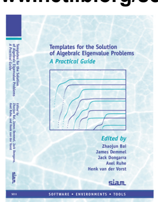




www.netlib.org/scalapack



gams.nist.gov



www.netlib.org/templates

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

A brief history of (Dense) Linear Algebra software (1/7)

- In the beginning was the do-loop…
  - Libraries like EISPACK (for eigenvalue problems)
- Then the BLAS (1) were invented (1973-1977)
  - Standard library of 15 operations (mostly) on vectors
    - “AXPY” (y = α·x + y), dot product, scale (x = α·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!)

Current Records for Solving Dense Systems (11/2015)

- **Linpack Benchmark**
- **Fastest machine overall (www.top500.org)**
  - **Tianhe-2 (Guangzhou, China)**
  - **33.9 Petaflops out of 54.9 Petaflops peak (n=10M)**
  - **3.1M cores, of which 2.7M are accelerator cores**
    - **Intel Xeon E5-2692 (Ivy Bridge) and Xeon Phi 31S1P**
  - **1 Pbyte memory**
  - **17.8 MWatts of power, 1.9 Gflops/Watt**

- **Historical data (www.netlib.org/performance)**
  - **Palm Pilot III**
  - **1.69 Kiloflops**
  - **n = 100**

A brief history of (Dense) Linear Algebra software (2/7)

- But the BLAS-1 weren't enough
  - Consider AXPY (y = α·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 (3/7)

- 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
    - Ex: 3 nested loops for GEMM
  - Lots more optimized code (eg Homework 1)
    - Motivates "automatic tuning" of the BLAS
  - Part of standard math libraries (eg AMD ACML, Intel MKL)

Level 1 BLAS

	dim scalar vector vector scalars 5-element array		prefixes
SUBROUTINE xROTG	(A, B, C, S)	Generate plane rotation	S, D
SUBROUTINE xROTMG	(D1, D2, A, B, PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT	(N, X, INCX, Y, INCY, C, S)	Apply plane rotation	S, D
SUBROUTINE xROTM	(N, X, INCX, Y, INCY, PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP	(N, X, INCX, Y, INCY)	$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL	(N, ALPHA, X, INCX)	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY	(N, X, INCX, Y, INCY)	$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY	(N, ALPHA, X, INCX, Y, INCY)	$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT	(N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU	(N, X, INCX, Y, INCY)	$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC	(N, X, INCX, Y, INCY)	$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT	(N, X, INCX, Y, INCY)	$dot \leftarrow \alpha + x^T y$	SDS
FUNCTION xNRM2	(N, X, INCX)	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
FUNCTION xASUM	(N, X, INCX)	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
FUNCTION IxAMAX	(N, X, INCX)	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

	options dim b-width scalar matrix vector scalar vector		
xGEMV	(TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xGBMV	(TRANS, M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
xHEMV	(U		
xHBMV	(U		
xHPMV	(U		
xSYMV	(U		
xSBMV	(U		
xSPMV	(U		
xTRMV	(U		
xTBMV	(U		
xTPMV	(U		
xTRSV	(U		
xTBSV	(U		
xTPSV	(U		
xGER	(		
xGERU	(		
xGERC	(		
xHER	(U		
xHPR	(U		
xHER2	(U		
xHPR2	(U		
xSYR	(U		
xSPR	(U		
xSYR2	(U		
xSPR2	(U		

BLAS Standards Committee to start meeting again May 2016:
Batched BLAS: many independent BLAS operations at once
Reproducible BLAS: getting bitwise identical answers from run-to-run, despite nonassociate floating point, and dynamic scheduling of resources (bebop.cs.berkeley.edu/reproblas)
Low-Precision BLAS: 16 bit floating point

See www.netlib.org/blas/blast-forum/ for previous extension attempt
New functions, Sparse BLAS, Extended Precision BLAS

Level 3

	options dim scalar matrix matrix scalar matrix		
xGEMM	(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha op(A) op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
xSYMM	(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
xHEMM	(SIDE, UPLO, M, N, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
xSYRK	(UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
xHERK	(UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
xSYR2K	(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
xHER2K	(UPLO, TRANS, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
xTRMM	(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A) B, B \leftarrow \alpha B op(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
xTRSM	(SIDE, UPLO, TRANSA, DIAG, M, N, ALPHA, A, LDA, B, LDB)	$B \leftarrow \alpha op(A^{-1}) B, B \leftarrow \alpha B op(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

2

A brief history of (Dense) Linear Algebra software (4/7)

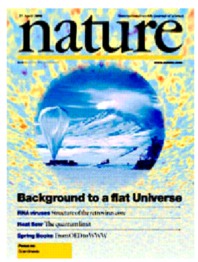
- LAPACK – "Linear Algebra PACKage" - uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1
    - How do we reorganize GE to use BLAS-3 ? (details later)
  - Contents of LAPACK (summary)
    - Algorithms that are (nearly) 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 = λ x
      - Singular Value Decomposition (SVD)
    - Generalized problems (eg Ax = λ Bx)
    - Error bounds for everything
    - Lots of variants depending on A's structure (banded, $A=A^T$, etc)
  - How much code? (Release 3.6.0, Nov 2015) (www.netlib.org/lapack)
    - Source: 1750 routines, 721K LOC, Testing: 1094 routines, 472K LOC
  - Ongoing development (at UCB and elsewhere) (class projects!)
    - Next planned release June 2016

A brief history of (Dense) Linear Algebra software (5/7)

- 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
    - Only subset of LAPACK's functionality available
    - Details later (class projects!)
  - All at www.netlib.org/scalapack

Success Stories for Sca/LAPACK (6/7)

- Widely used
  - Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, …
  - 7.5M webhits/year @ Netlib (incl. CLAPACK, LAPACK95)
- New Science discovered through the solution of dense matrix systems
  - Nature article on the flat universe used ScaLAPACK
  - Other articles in Physics Review B that also use it
  - 1998 Gordon Bell Prize
  - www.nersc.gov/assets/NewsImages/2003/newNERSCresults050703.pdf



Cosmic Microwave Background Analysis, BOOMERanG collaboration, MADCAP code (Apr. 27, 2000).

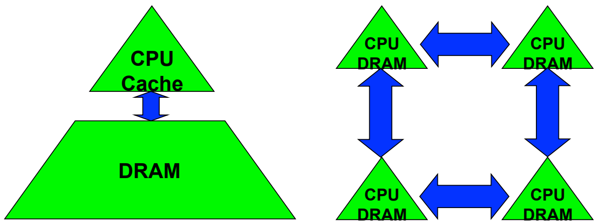
A brief future look at (Dense) Linear Algebra software (7/7)

- 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?
  - Details later (Class projects!) (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
- So far, none of these libraries minimize communication in all cases (not even matmul!)

Back to basics:
Why avoiding communication is important (1/3)

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 (2/3)

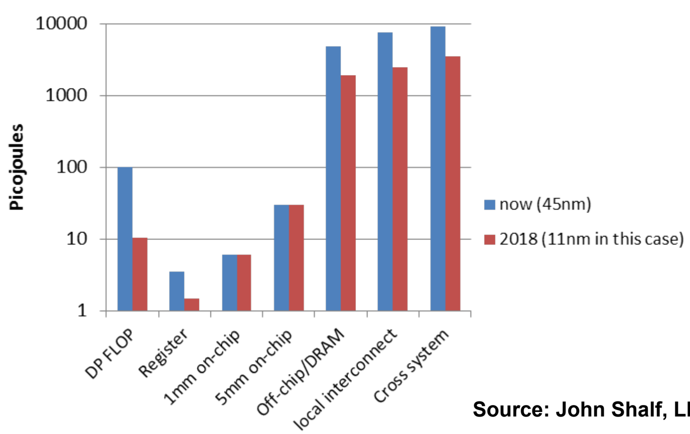
- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

  (# words moved / bandwidth and # messages * latency: **communication**)
- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	DRAM	26%	15%
	Network	23%	5%

- **Minimize communication to save time**

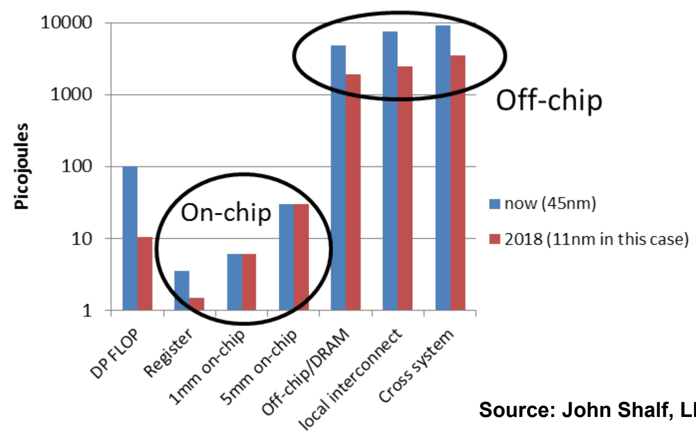
Why Minimize Communication? (3/3)



Source: John Shalf, LBL

Why Minimize Communication? (3/3)

Minimize communication to save energy



Source: John Shalf, LBL

Goal:
Organize Linear Algebra to Avoid Communication

- Between all memory hierarchy levels
  - L1 ⟷ L2 ⟷ DRAM ⟷ network, etc
- Not just *hiding* communication (overlap with arithmetic)
  - Speedup ≤ 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·B breaks A, B and C into blocks with dimensions that depend on cache size

… Break A^{nxn}, B^{nxn}, C^{nxn} into bxb blocks labeled A(i,j), etc
… b chosen so 3 bxb 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)·B(k,j) … b x 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 = $\Omega(n^3 / M^{1/2})$ [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 $= \Omega((n^3/p) / M^{1/2})$ [Irony, Tiskin, Toledo, 04]
  - If $M = 3n^2/p$ (one copy of each matrix), then lower bound $= \Omega(n^2/p^{1/2})$
  - 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 = $\Omega(\#flops / M^{1/2})$

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

- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
  - Some whole programs (sequences of these operations, no matter how they are interleaved, eg computing A^k)
  - Dense *and* sparse matrices (where #flops << n^3)
  - Sequential *and* parallel algorithms
  - Some graph-theoretic algorithms (eg 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 / $M^{1/2}$)

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

- Sequential case, dense n x 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 x n matrices
  - 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 = $\Omega(p^{1/2})$

SIAM Linear Algebra Prize, 2012

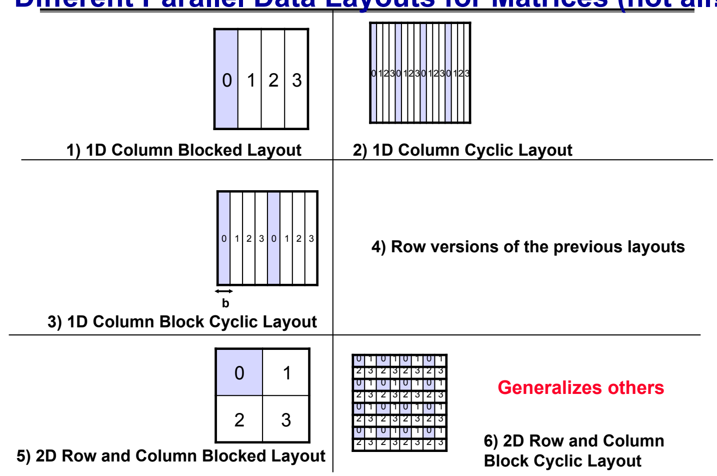
Can we attain these lower bounds?

- 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
    - Need new data structures
  - Minimize for multiple memory hierarchy levels
    - Cache-oblivious algorithms would be simplest
  - Fewest flops when matrix fits in fastest memory
    - Cache-oblivious algorithms don't always attain this
- Attainable for nearly all dense linear algebra
  - Just a few prototype implementations so far (class projects!)
  - Only a few sparse algorithms so far (eg Cholesky)

Outline

- History and motivation
  - What is dense linear algebra?
  - Why minimize communication?
  - Lower bound on communication
- Parallel Matrix-matrix multiplication
  - Attaining the lower bound
- Other Parallel Algorithms (next lecture)

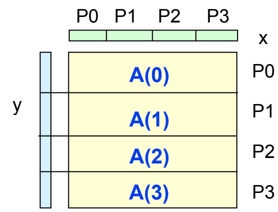
Different Parallel Data Layouts for Matrices (not all!)



Parallel Matrix-Vector Product

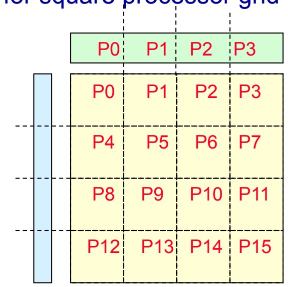
- Compute y = y + A*x, where A is a dense matrix
- Layout:
  - **1D row blocked**
- A(i) refers to the n by n/p block row that processor i owns,
- x(i) and y(i) similarly refer to segments of x,y owned by i
- **Algorithm:**
  - **Foreach processor i**
  - **Broadcast x(i)**
  - **Compute y(i) = A(i)*x**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

  $y(i) = y(i) + A(i)*x = y(i) + \Sigma_j A(i,j)*x(j)$



Matrix-Vector Product y = y + A*x

- A column layout of the matrix eliminates the broadcast of x
  - But adds a reduction to update the destination y
- A 2D blocked layout uses a broadcast and reduction, both on a subset of processors
  - sqrt(p) for square processor grid



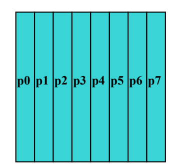
Parallel Matrix Multiply

- Computing C=C+A*B
- Using basic algorithm: $2*n^3$ 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 * parallel time)
  - perfect (linear) speedup ↔ efficiency = 1

Matrix Multiply with 1D Column Layout

- Assume matrices are n x 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(i,j) is the n/p by n/p sublock of B(i)
  - in rows j*n/p through (j+1)*n/p - 1
- Algorithm uses the formula

  $C(i) = C(i) + A*B(i) = C(i) + \Sigma_j A(j)*B(j,i)$

Matrix Multiply: 1D Layout on Bus or Ring

- Algorithm uses the formula
  $C(i) = C(i) + A*B(i) = C(i) + \Sigma_j A(j)*B(j,i)$
- 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 without Broadcast

Naïve algorithm:

```
C(myproc) = C(myproc) + A(myproc)*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)*B(i,myproc)
    barrier
```

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2 /p$

Naïve MatMul (continued)

Cost of inner loop:

computation: $2*n*(n/p)^2 = 2*n^3/p^2$

communication: $\alpha + \beta*n^2 /p$ … approximately

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:

$= (p*(p-1) + 1)*\text{computation} + p*(p-1)*\text{communication}$

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

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**

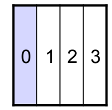
```
Copy A(myproc) into Tmp
C(myproc) = C(myproc) + Tmp*B(myproc , myproc)
for j = 1 to p-1
    Send Tmp to processor myproc+1 mod p
    Receive Tmp from processor myproc-1 mod p
    C(myproc) = C(myproc) + Tmp*B( myproc-j mod p , myproc)
```

- **Same idea as for gravity in simple sharks and fish algorithm**
  - **May want double buffering in practice for overlap**
  - **Ignoring deadlock details in code**
- **Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$**

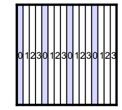
Matmul for 1D layout on a Processor Ring

- Time of inner loop = $2*(\alpha + \beta*n^2/p) + 2*n*(n/p)^2$
- Total Time = $2*n*(n/p)^2 + (p-1) *$ Time of inner loop
- $\approx 2*n^3/p + 2*p*\alpha + 2*\beta*n^2$
- (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)*cost of sending n*(n/p) words
- Parallel Efficiency = $2*n^3$ / (p * Total Time)
  $= 1/(1 + \alpha * p^2/(2*n^3) + \beta * p/(2*n))$
  $= 1/(1 + O(p/n))$
- Grows to 1 as n/p increases (or α and β shrink)
- But far from communication lower bound

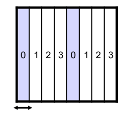
Need to try 2D Matrix layout



1) 1D Column Blocked Layout

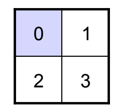


2) 1D Column Cyclic Layout

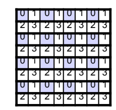


3) 1D Column Block Cyclic Layout

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 Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Presentation from van de Geijn and Watts
  - www.netlib.org/lapack/lawns/lawn96.ps
  - Similar ideas appeared many times
- Used in practice in PBLAS = Parallel BLAS
  - www.netlib.org/lapack/lawns/lawn100.ps

SUMMA uses Outer Product form of MatMul

- C = A*B means $C(i,j) = \Sigma_k A(i,k)*B(k,j)$
- Column-wise outer product:

  C = A*B

  $= \Sigma_k A(:,k)*B(k,:)$

  $= \Sigma_k$ (k-th col of A)*(k-th row of B)

- Block column-wise outer product (block size = 4 for illustration)

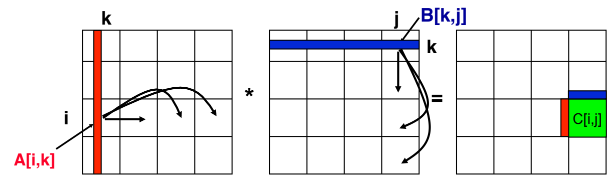
  C = A*B

  = A(:,1:4)*B(1:4,:) + A(:,5:8)*B(5:8,:) + …

  $= \Sigma_k$ (k-th block of 4 cols of A)* (k-th block of 4 rows of B)

02/25/2016 CS267 Lecture 12 41

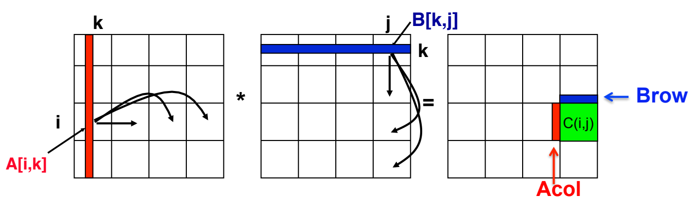
SUMMA – n x n matmul on $P^{1/2}$ x $P^{1/2}$ grid



- C[i, j] is $n/P^{1/2}$ x $n/P^{1/2}$ submatrix of C on processor P_{ij}
- A[i,k] is $n/P^{1/2}$ x b submatrix of A
- B[k,j] is b x $n/P^{1/2}$ submatrix of B
- $C[i,j] = C[i,j] + \Sigma_k A[i,k]*B[k,j]$
  - summation over submatrices
- Need not be square processor grid

02/25/2016 CS267 Lecture 12 42

SUMMA– n x n matmul on $P^{1/2}$ x $P^{1/2}$ 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 bin. tree)
   Receive A[i,k] into Acol
   Receive B[k,j] into Brow
   C_myproc = C_myproc + Acol * Brow
```

02/25/2016 CS267 Lecture 12 43

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

44

Performance of PBLAS

PDGEMM = PBLAS routine for matrix multiply

Observations:
For fixed N, as P increases Mflops increases, but less than 100% efficiency
For fixed P, as N increases, Mflops (efficiency) rises

Speed in Mflops of PDGEMM

Machine	Procs	Block Size	N = 2000	N = 4000	N = 10000
Cray T3E	4=2x2	32	1055	1070	0
	16=4x4		3630	4005	4292
	64=8x8		13456	14287	16755
IBM SP2	4	50	755	0	0
	16		2514	2850	0
	64		6205	8709	10774
Intel XP/S MP Paragon	4	32	330	0	0
	16		1233	1281	0
	64		4496	4864	5257
Berkeley NOW	4	32	463	470	0
	32=4x8		2490	2822	3450
	64		4130	5457	6647

DGEMM = BLAS routine for matrix multiply

Maximum speed for PDGEMM = # Procs * speed of DGEMM

Observations (same as above):
Efficiency always at least 48%
For fixed N, as P increases, efficiency drops
For fixed P, as N increases, efficiency increases

Efficiency = MFlops(PDGEMM)/(Procs*MFlops(DGEMM))

Machine	Peak/proc	DGEMM Mflops	Procs	N = 2000	N = 4000	N = 10000
Cray T3E	600	360	4	.73	.74	
			16	.63	.70	.75
			64	.58	.62	.73
IBM SP2	266	200	4	.94		
			16	.79	.89	
			64	.48	.68	.84
Intel XP/S MP Paragon	100	90	4	.92		
			16	.86	.89	
			64	.78	.84	.91
Berkeley NOW	334	129	4	.90	.91	
			32	.60	.68	.84
			64	.50	.66	.81

02/25/2016 45

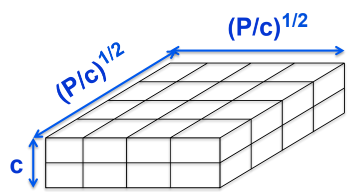
Can we do better?

- 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 = Ω(#flops / $M^{1/2}$) = Ω(n^2 / ($c^{1/2}$ $P^{1/2}$))
  - #messages = Ω(#flops / $M^{3/2}$) = Ω($P^{1/2}$ / $c^{3/2}$)
- 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 $P^{1/3}$ x $P^{1/3}$ x $P^{1/3}$ grid
    - Processor (i,j,k) performs C(i,j) = C(i,j) + A(i,k)*B(k,j), where each submatrix is n/$P^{1/3}$ x n/$P^{1/3}$
  - Not always that much memory available…

02/25/2016 CS267 Lecture 12 46

2.5D Matrix Multiplication

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

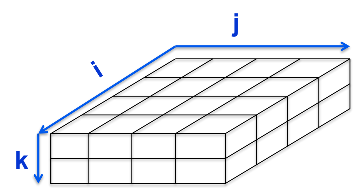


Example: P = 32, c = 2

02/25/2016 CS267 Lecture 12

2.5D Matrix Multiplication

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

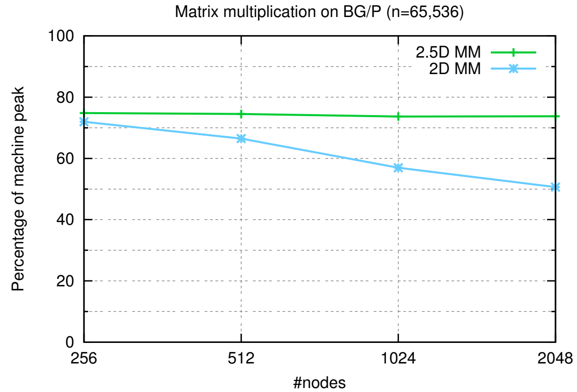


Initially P(i,j,0) owns A(i,j) and B(i,j) each of size $n(c/P)^{1/2}$ x $n(c/P)^{1/2}$

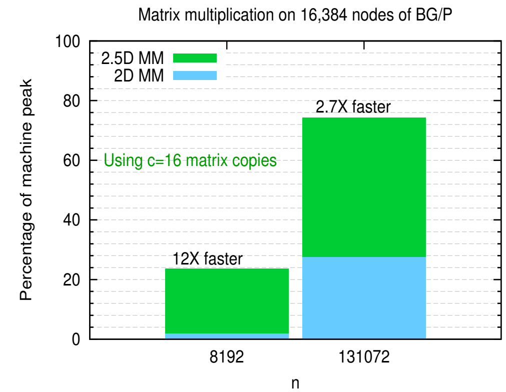
(1) P(i,j,0) broadcasts A(i,j) and B(i,j) to P(i,j,k)
(2) Processors at level k perform 1/c-th of SUMMA, i.e. 1/c-th of Σ_m A(i,m)*B(m,j)
(3) Sum-reduce partial sums Σ_m A(i,m)*B(m,j) along k-axis so P(i,j,0) owns C(i,j)

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 = Ω(#flops / $M^{1/2}$) = Ω(n^2 / ($c^{1/2} P^{1/2}$))
  - #messages = Ω(#flops / $M^{3/2}$) = Ω($P^{1/2} / c^{3/2}$)
- Perfect strong scaling! But only up to c = $P^{1/3}$

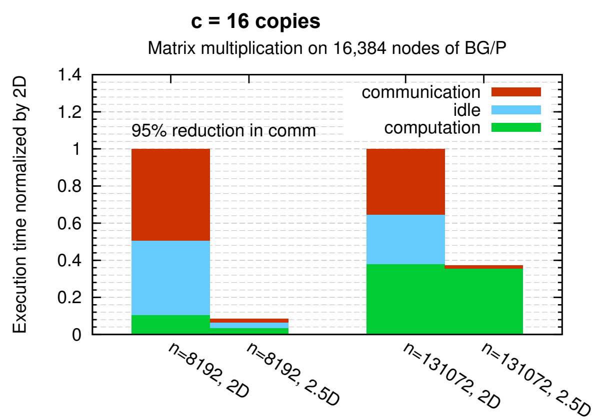


2.5D Matmul on IBM BG/P, 16K nodes / 64K cores



02/25/2016 CS267 Lecture 12

2.5D Matmul on IBM BG/P, 16K nodes / 64K cores



02/25/2016

Distinguished Paper Award, EuroPar'11
SC'11 paper by Solomonik, Bhatele, D.

Perfect Strong Scaling – in Time and Energy

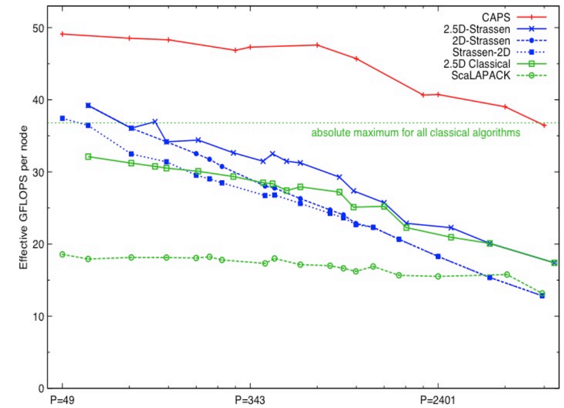
- Every time you add a processor, you should use its memory M too
- Start with minimal number of procs: $PM = 3n^2$
- Increase P by a factor of c ➔ total memory increases by a factor of c
- Notation for timing model:
  - γ_T , β_T , α_T = secs per flop, per word_moved, per message of size m
- $T(cP) = n^3/(cP) [\gamma_T + \beta_T/M^{1/2} + \alpha_T/(mM^{1/2})]$
  $= T(P)/c$
- Notation for energy model:
  - γ_E , β_E , α_E = joules for same operations
  - δ_E = joules per word of memory used per sec
  - ε_E = joules per sec for leakage, etc.
- $E(cP) = cP \{ n^3/(cP) [\gamma_E + \beta_E/M^{1/2} + \alpha_E/(mM^{1/2})] + \delta_E M T(cP) + \varepsilon_E T(cP) \}$
  $= E(P)$
- c cannot increase forever: c <= $P^{1/3}$ (3D algorithm)
  - Corresponds to lower bound on #messages hitting 1
- Perfect scaling extends to Strassen's matmul, direct N-body, …
  - "Perfect Strong Scaling Using No Additional Energy"
  - "Strong Scaling of Matmul and Memory-Indep. Comm. Lower Bounds"
  - Both at bebop.cs.berkeley.edu

Classical Matmul

- 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 $\Omega(n/p^{2/3})$

Strong scaling of Matmul on Hopper (n=94080)

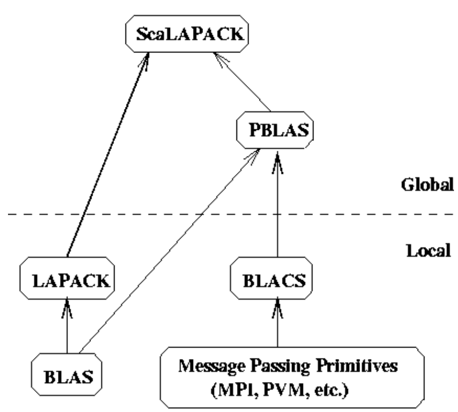
G. Ballard, D., O. Holtz, B. Lipshitz, O. Schwartz



"Communication-Avoiding Parallel Strassen"
bebop.cs.berkeley.edu, Supercomputing'12

ScaLAPACK Parallel Library

ScaLAPACK SOFTWARE HIERARCHY



Extensions of Lower Bound and Optimal Algorithms

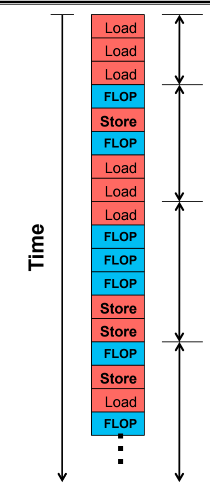
- For each processor that does G flops with fast memory of size M
  #words_moved = $\Omega(G/M^{1/2})$
- Extension: for any program that "smells like"
  - Nested loops …
  - That access arrays …
  - Where array subscripts are linear functions of loop indices
    - Ex: A(i,j), B(3*i-4*k+5*j, i-j, 2*k, …), …
  - There is a constant s such that
    #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 …

Proof of Communication Lower Bound on C = A·B (1/4)

- Proof from Irony/Toledo/Tiskin (2004)
- Think of instruction stream being executed
  - Looks like " … add, load, multiply, store, load, add, …"
    - Each load/store moves a word between fast and slow memory
  - We want to count the number of loads and stores, given that we are multiplying n-by-n matrices C = A·B using the usual $2n^3$ flops, possibly reordered assuming addition is commutative/associative
  - Assuming that at most M words can be stored in fast memory
- Outline:
  - Break instruction stream into segments, each with M loads and stores
  - Somehow bound the maximum number of flops that can be done in each segment, call it F
  - So $F \cdot \#\text{ segments} \geq T = \text{total flops} = 2 \cdot n^3$, so $\#\text{ segments} \geq T / F$
  - So $\#\text{ loads \& stores} = M \cdot \#\text{segments} \geq M \cdot T / F$

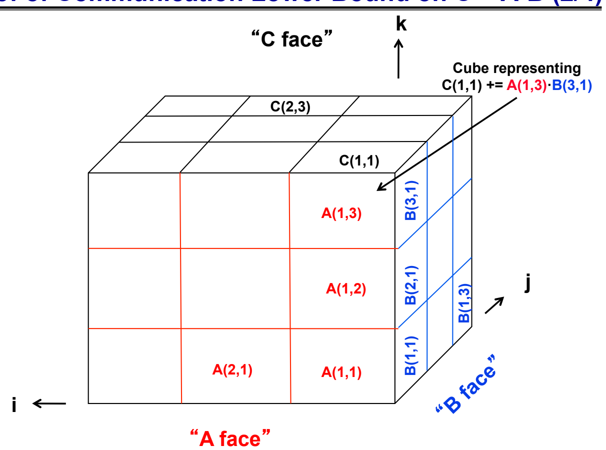
02/25/2016 CS267 Lecture 12 57

Illustrating Segments, for M=3



02/25/2016 58

Proof of Communication Lower Bound on C = A·B (2/4)



- If we have at most 2M "A squares", 2M "B squares", and 2M "C squares" on faces, how many cubes can we have?

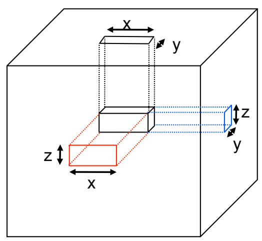
59

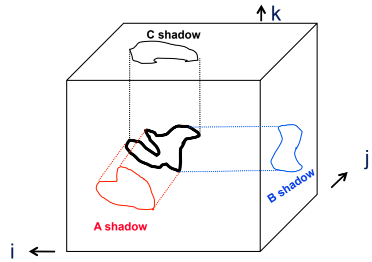
Proof of Communication Lower Bound on C = A·B (3/5)

- Given segment of instruction stream with M loads & stores, how many adds & multiplies (F) can we do?
  - At most 2M entries of C, 2M entries of A and/or 2M entries of B can be accessed
- Use geometry:
  - Represent n^3 multiplications by n x n x n cube
  - One n x n face represents A
    - each 1 x 1 subsquare represents one A(i,k)
  - One n x n face represents B
    - each 1 x 1 subsquare represents one B(k,j)
  - One n x n face represents C
    - each 1 x 1 subsquare represents one C(i,j)
  - Each 1 x 1 x 1 subcube represents one C(i,j) += A(i,k) · B(k,j)
    - May be added directly to C(i,j), or to temporary accumulator

60

Proof of Communication Lower Bound on C = A·B (3/4)





 # cubes in black box with side lengths x, y and z
= Volume of black box
= x·y·z
= (xz · zy · yx)$^{1/2}$
= (#A□s · #B□s · #C□s)$^{1/2}$

(i,k) is in A shadow if (i,j,k) in 3D set
(j,k) is in B shadow if (i,j,k) in 3D set
(i,j) is in C shadow if (i,j,k) in 3D set

Thm (Loomis & Whitney, 1949)
 # cubes in 3D set = Volume of 3D set
≤ (area(A shadow) · area(B shadow) · area(C shadow)) $^{1/2}$

61

Proof of Communication Lower Bound on C = A·B (4/4)

- Consider one "segment" of instructions with M loads, stores
- Can be at most 2M entries of A, B, C available in one segment
- Volume of set of cubes representing possible multiply/adds in one segment is ≤ (2M · 2M · 2M)$^{1/2}$ = (2M) $^{3/2}$ ≡ F
- # Segments ≥ $\lfloor 2n^3 / F \rfloor$
- # Loads & Stores = M · #Segments ≥ M · $\lfloor 2n^3 / F \rfloor$
  ≥ $n^3 / (2M)^{1/2} - M = \Omega(n^3 / M^{1/2})$

- Parallel Case: apply reasoning to one processor out of P
  - # Adds and Muls ≥ $2n^3$ / P (at least one proc does this)
  - M= n^2 / P (each processor gets equal fraction of matrix)
  - # "Load & Stores" = # words moved from or to other procs
    ≥ M · ($2n^3$ /P) / F= M · ($2n^3$ /P) / $(2M)^{3/2}$ = $n^2 / (2P)^{1/2}$

62