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 $\pi$ …
  • That has a race condition, so that it gives you a different funny answer every time you run it?

Monte - $\pi$ - 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
- Eigenvales and vectors of Symmetric Matrices
  - Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)
- Eigenvales and vectors of Unsymmetric matrices
  - Eigenvales, 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\backslash 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
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" ($y = \alpha \cdot x + y$), 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 (11/2013)

- 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 M Watts 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 = α·A·x + β·y, “GER”: A = A + α·x·yᵀ, x = T⁻¹·x
    - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC
  - Why BLAS 2? They do O(n²) ops on O(n²) data
  - So computational intensity still just ~((2n²)/(n²)) = 2
    - OK for vector machines, but not for machine with caches

  - Standard library of 9 operations (mostly) on matrix/matrix pairs
    - “GEMM”: C = α·A·B + β·C, C = α·A·Aᵀ + β·C, B = T⁻¹·B
    - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3? They do O(n³) ops on O(n²) data
  - So computational intensity (2n³)/(4n²) = 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)

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
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 \(\approx 50\%\) BLAS 3
    – Eigenproblems: Find \(\lambda\) and x where \(Ax = \lambda x\)
    – Singular Value Decomposition (SVD)
  • Generalized problems (eg Ax = \(\lambda\) Bx)
  • Error bounds for everything
  • Lots of variants depending on \(A\text{’s structure} \ (\text{banded, } A=\text{A}^T, \text{etc})\)
• 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

Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1

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
  • Time_{per_flop} << 1/ bandwidth << latency
• Gaps growing exponentially with time

Why Minimize Communication? (3/3)

Minimize communication to save time

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 \( \leq 2x \)
- 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
  - \( \ldots \) Break \( A^{m \times n}, B^{m \times n}, C^{n \times n} \) 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 \( n/b \)
    \[
    C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)
    \]
  - \( b \) x b matmul, \( 4b^2 \) reads/writes
  - When \( b=1 \), get “naïve” algorithm, want \( b \) larger ...
  - \( (\frac{n}{b})^3 \cdot 4b^2 = 4n^3/b \) reads/writes altogether
  - Minimized when \( 3b^2 = \) cache size = \( M \), yielding \( O(n^3/M^{1/2}) \) reads/writes

---

**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
    \[
    = \Omega \left( \frac{(n^3/p)}{M^{1/2}} \right) \]
    [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

\[
\text{#words\_moved per processor} = \Omega(\#flops / M^{1/2})
\]
\[
\text{#messages\_sent per processor} = \Omega(\#flops / M^{3/2})
\]

- 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 \( A^3 \))
  - Dense and sparse matrices (where \#flops \( \ll n^3 \))
  - 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 = $\Omega (#flops / M^{1/2})$
#messages\_sent per processor = $\Omega (#flops / M^{3/2})$

- Sequential case, dense $n \times n$ matrices, so $O(n^3)$ flops
  - #words\_moved = $\Omega(n^2 / M^{1/2})$
  - #messages\_sent = $\Omega(n^2 / M^{3/2})$
- Parallel case, dense $n \times n$ 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 = $\Omega(p^{1/2})$

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!)
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
6) 2D Row and Column Block Cyclic Layout
Generalizes others
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:
    $$y(i) = y(i) + A(i)*x = y(i) + \sum_j A(i,j)*x(j)$$

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

Matrix Multiply with 1D Column Layout

- Assume matrices are $n \times n$ and $n$ is divisible by $p$
- **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** = \textit{``latency''} + \#words \times time-per-word
    $$= \alpha + n \beta$$
  - Efficiency (in any model):
    - serial time / (p \times parallel time)
    - perfect (linear) speedup $\leftrightarrow$ efficiency $= 1$
  - A(i) refers to the $n$ by $n/p$ block column that processor $i$ owns (similarly for $B(i)$ and $C(i)$)
    - $B(i,j)$ is the $n/p$ by $n/p$ subblock of $B(i)$
      - in rows $j*n/p$ through $(j+1)*n/p - 1$
  - Algorithm uses the formula
    $$C(i) = C(i) + A^*B(i) = C(i) + \sum_j A(j)^*B(j,i)$$

May be a reasonable assumption for analysis, not for code
Matrix Multiply: 1D Layout on Bus or Ring

- Algorithm uses the formula
  \[ C(i) = C(i) + A \times B(i) = C(i) + \sum_j A(j) \times B(j,i) \]

- 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

Naïve MatMul (continued)

Cost of inner loop:
- computation: \( 2 \times n \times (n/p)^2 \)
- communication: \( \alpha + \beta \times n^3 / p \) ... 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:

\[
= (p \times (p-1) + 1) \times \text{computation} + p \times (p-1) \times \text{communication} \\
= 2 \times n^3 + p^2 \alpha + p^2 n^2 \beta
\]

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

<table>
<thead>
<tr>
<th>Copy A(myproc) into Tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(myproc) = C(myproc) + Tmp \times B(myproc, myproc)</td>
</tr>
<tr>
<td>for j = 1 to p-1</td>
</tr>
<tr>
<td>Send Tmp to processor myproc+1 mod p</td>
</tr>
<tr>
<td>Receive Tmp from processor myproc-1 mod p</td>
</tr>
<tr>
<td>C(myproc) = C(myproc) + Tmp \times B( myproc-j mod p , myproc)</td>
</tr>
</tbody>
</table>

- 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 = \( 2 \times (\alpha + \beta \times n^5 / p) + 2 \times n^3 / (n/p)^2 \)
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
- $= 2^*n^2/p + 2^*n^2 + 2^*\beta n^2$
- (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 $\alpha$ and $\beta$ shrink)
- But far from communication lower bound

Need to try 2D Matrix layout

1) 1D Column Blocked Layout
2) 1D Column Cyclic Layout
4) Row versions of the previous layouts
3) 1D Column Block Cyclic Layout
5) 2D Row and Column Blocked Layout
6) 2D Row and Column Block Cyclic Layout

Generalizes others

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:**
  \[
  C = A*B = \Sigma_k (k-th block of 4 cols of A) \times (k-th block of 4 rows of B)
  \]

- **Block column-wise outer product (block size = 4 for illustration):**
  \[
  C = A*B = A(:,1:4)*B(1:4,:) + A(:,5:8)*B(5:8,:) + \ldots
  \]
  \[
  = \Sigma_k (k-th block of 4 cols of A) \times (k-th block of 4 rows of B)
  \]

---

**SUMMA – n x 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 \( B \)
- \( C[i,j] = C[i,j] + \Sigma_k A[i,k]*B[k,j] \)
- summation over submatrices
- Need not be square processor grid

---

**SUMMA Costs**

For \( k=0 \) to \( n/b-1 \)
- for all \( i = 1 \) to \( P^{1/2} \)
  - \( C[I,k] \) 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 \( P^{1/2} \)
  - \( C[k,J] \) broadcasts it to whole processor column (using binary tree)
  - \#words and \#messages
  - Receive \( C[I,k] \) into \( Acol \)
  - Receive \( C[k,J] \) into \( Brow \)
  - \( C_{myproc} = C_{myproc} + Acol \times Brow \)
  - \#flops = \( 2n^2b/P \)

- 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 = \( 2n^3/P \)
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

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

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}
    • Bernstein 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...

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

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

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

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 \( \Rightarrow \) total memory increases by a factor of c
- Notation for timing model:
  - \( Y_1, \beta_1, \alpha_c = \text{secs per flop, per word_moved, per message of size m} \)
  - \( T(cP) = n^3/cP [ Y_1 + \beta_1/M^{1/2} + \alpha_c/(mM^{1/2}) ] \)
  - \( T(P)/c \)
- Notation for energy model:
  - \( Y_E, \beta_E, \alpha_E = \text{joules for same operations} \)
  - \( \delta_E = \text{joules per word of memory used per sec} \)
  - \( \epsilon_E = \text{joules per sec for leakage, etc.} \)
  - \( E(cP) = cP \{ \text{flops} [ Y_E + \beta_E/M^{1/2} + \alpha_E/(mM^{1/2}) ] + \delta_E MT(cP) + \epsilon_E T(cP) \} \)
  - \( E(P) \)
- \( c \) cannot increase forever: \( c \leq P^{1/3} \) (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:
  $$\Omega\left((n^3/p)/M^{1/2}\right) = \Omega\left((n/M^{1/2})^3/p\right)$$
- Communication lower bound on #messages:
  $$\Omega\left((n^3/p)/M^{3/2}\right) = \Omega\left((n/M^{1/2})^3/p\right)$$
- 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

```
bbebop.cs.berkeley.edu, Supercomputing’12
```

### ScaLAPACK Parallel Library

#### ScaLAPACK SOFTWARE HIERARCHY

- 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 · # segments ≥ T = total flops = 2 · n^3, so # segments ≥ T / F
- So # loads & stores = M · #segments ≥ M · T / F

Illustrating Segments, for M=3

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?

Cube representing C(1,1) += A(1,3)·B(3,1)

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

- 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 \( \leq (2M \cdot 2M \cdot 2M)^{1/2} = (2M)^{3/2} \equiv F \).
- \# Segments \( \geq \lceil 2n^3 / F \rceil \).
- \# Loads & Stores \( = M \cdot \# \text{Segments} \geq M \cdot \lceil 2n^3 / F \rceil \)
  \[ \geq n^3 / (2M)^{1/2} - M = \Omega(n^3 / M^{1/2}) \]

• Parallel Case: apply reasoning to one processor out of \( P \):
  • \# Adds and Muls \( \geq 2n^3 / P \) (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
    \[ \geq M \cdot (2n^3 / P) / F = M \cdot (2n^3 / P) / (2M)^{3/2} = n^2 / (2P)^{1/2} \]