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
  • Why minimize communication?
  • Lower bound on communication
• Structure of the Dense Linear Algebra motif
  • What does $A\backslash b$ do?
• Parallel Matrix-matrix multiplication
  • Attaining the lower bound
  • Proof of the lower bound (if time)
• Parallel Gaussian Elimination (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, weighted
• Eigenvectors and vectors of Symmetric Matrices
  • Standard \((A = kI)\), Generalized \((A = kB)\)
• Eigenvectors 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…)

Current Records for Solving Dense Systems (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
• 3.1M cores, of which 2.7M are accelerator cores
• 1 Pbyte memory
• 17.8 MWatts of power, 1.9 Gflops/Watt

• Historical data (www.netlib.org/performance)
  • Palm Pilot III
  • 1.69 Kiloflops
  • \( n = 100 \)
Current Records for Solving Dense Systems (11/2012)

- Linpack Benchmark
- Fastest machine overall (www.top500.org)
  - Cray TITAN (Oak Ridge National Lab)
  - 17.6 Petaflops out of 27.1 Petaflops peak
  - 18,688 compute nodes, each with 16 core Opteron and Nvidia Tesla K20 GPU
  - 299,008 Opteron cores
  - 710 Terabytes memory
  - 8.2 MW of power

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


- Linpack Benchmark
- Fastest machine overall (www.top500.org)
  - Fujitsu K-Computer (RIKEN Institute, Japan)
  - 10.5 Petaflops out of 11.3 Petaflops peak
  - n = 11.9M, 29.5 hours to run
  - 705K cores, 12.7 MW of power

- 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·x
    - “GER”: A = A + α·x·y
    - “TRSV”: y = 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

A brief history of (Dense) Linear Algebra software (3/7)

  - Standard library of 9 operations (mostly) on matrix/matrix pairs
    - “GEMM”: C = α·A·B + β·C, C = α·A·Aᵀ + β·C, C = 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(n⁳) data
  - So computational intensity (2n³)/(4n³) = 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)
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 (small) subset of LAPACK’s functionality available
    - Details later (class projects!)
  - All at www.netlib.org/scalapack

Success Stories for Sc/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/news/reports/newNERSCresults050703.pdf

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

  - 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 \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’s structure (banded, A=A^T, etc)
- How much code? (Release 3.5.0, Nov 2013) (www.netlib.org/lapack)
  - Source: 1740 routines, 704K LOC, Testing: 1096 routines, 467K LOC
  - Ongoing development (at UCB and elsewhere) (class projects!)
  - Next planned release Mar 2014
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/{{dplasma,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, ...

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

<table>
<thead>
<tr>
<th>Annual improvements</th>
<th>Time_per_flop</th>
<th>Bandwidth</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>59%</td>
<td>26%</td>
<td>15%</td>
</tr>
<tr>
<td></td>
<td>23%</td>
<td>5%</td>
<td></td>
</tr>
</tbody>
</table>

• Minimize communication to save time

Why Minimize Communication? (3/3)

Source: John Shalf, LBL
**Why Minimize Communication? (3/3)**

Minimize communication to save energy

- On-chip
- Off-chip

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

  ... Break $A^{bn}$, $B^{bn}$, $C^{bn}$ into $b \times b$ blocks labeled $A(i,j)$, etc

  ... $b$ chosen so $3 \times b$ 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 "naive" algorithm, want $b$ larger ...
- $(n/b)^3 \cdot 4b^2 = 4n^3/b$ reads/writes altogether
- Minimized when $3b^2 = \text{cache size} = M$, 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
    - $= \Omega (n^3 / (p \cdot M^{1/2}))$ [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, 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)

• Proof later, if time

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 $#\text{words\_moved}$
  • Minimize $#\text{messages\_sent}$
  • 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 (e.g., Cholesky)

Outlook

• History and motivation
  • Why minimize communication?
  • Lower bound on communication
• Structure of the Dense Linear Algebra motif
  • What does $A\backslash b$ do?
  • Parallel Matrix-matrix multiplication
  • Attaining the lower bound
  • Proof of the lower bound (if time)
  • Parallel Gaussian Elimination (next lecture)
What could go into the linear algebra motif(s)?

For all linear algebra problems
For all matrix/problem structures
For all data types
For all architectures and networks
For all programming interfaces

Produce best algorithm(s) w.r.t. performance and/or accuracy
(including error bounds, etc)

Need to prioritize, automate!

For all linear algebra problems:
Ex: LAPACK Table of Contents

- Linear Systems
  - Least Squares
    - Overdetermined, underdetermined
    - Unconstrained, constrained, weighted
- Eigenvalues and vectors of Symmetric Matrices
  - Standard (Ax = λx), Generalized (Ax = λBx)
- Eigenvalues and vectors of Unsymmetric matrices
  - Eigenvectors, Schur form, invariant subspaces
  - Standard, Generalized
- Singular Values and vectors (SVD)
  - Standard, Generalized
- Level of detail
  - Simple Driver
  - Expert Drivers with error bounds, extra-precision, other options
  - Lower level routines ("apply certain kind of orthogonal transformation")

What does A\b do? What could it do?
Ex: LAPACK Table of Contents

- BD – bidiagonal
- GB – general banded
- GE – general
- GG – general, pair
- GT – tridiagonal
- HB – Hermitian banded
- HE – Hermitian
- HG – upper Hessenberg, pair
- HP – Hermitian, packed
- HS – upper Hessenberg
- OR – (real) orthogonal
- OP – (real) orthogonal, packed
- PB – positive definite, banded
- PO – positive definite
- PP – positive definite, packed
- PT – positive definite, tridiagonal

- SB – symmetric, banded
- SP – symmetric, packed
- ST – symmetric, tridiagonal
- SY – symmetric
- TB – triangular, banded
- TG – triangular, pair
- TP – triangular, packed
- TR – triangular
- TZ – trapezoidal
- UN – unitary
- UP – unitary packed

- SB – symmetric, banded
- SP – symmetric, packed
- ST – symmetric, tridiagonal
- SY – symmetric
- TB – triangular, banded
- TG – triangular, pair
- TP – triangular, packed
- TR – triangular
- TZ – trapezoidal
- UN – unitary
- UP – unitary packed

02/27/2014 CS267 Lecture 12
02/27/2014 CS267 Lecture 12
What does A \backslash b do? What could it do?

Ex: LAPACK Table of Contents

• BD – bidiagonal
• GB – general banded
• GE – general
• GG – general, pair
• GT – tridiagonal
• HB – Hermitian banded
• HE – Hermitian
• HG – upper Hessenberg, pair
• HP – Hermitian, packed
• HS – upper Hessenberg
• OR – (real) orthogonal
• OP – (real) orthogonal, packed
• PB – positive definite, banded
• PO – positive definite
• PP – positive definite, packed
• PT – positive definite, tridiagonal

• SB – symmetric, banded
• SP – symmetric, packed
• ST – symmetric, tridiagonal
• SY – symmetric
• TB – triangular, banded
• TG – triangular, pair
• TP – triangular, packed
• TR – triangular
• TZ – trapezoidal
• UN – unitary
• UP – unitary packed

Organizing Linear Algebra – in books

www.netlib.org/lapack
www.netlib.org/scalapack
www.cs.utk.edu/~dongarra/etemplates
www.gams.nist.gov
Outline

- History and motivation
  - Why minimize communication?
  - Lower bound on communication
- Structure of the Dense Linear Algebra motif
  - What does \( A \backslash b \) do?
- Parallel Matrix-matrix multiplication
  - Attaining the lower bound
  - Proof of the lower bound (if time)
- Parallel Gaussian Elimination (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/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:
  - 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
Parallel Matrix Multiply

- Computing \( C = C + A \times B \)
- Using basic algorithm: \( 2 \times n^3 \) 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 \( n \times n \) and \( n \) is divisible by \( p \)

MatMul: 1D layout on Bus without Broadcast

Naïve algorithm:

\[
C(\text{myproc}) = C(\text{myproc}) + A(\text{myproc}) \times B(\text{myproc,myproc})
\]

for \( i = 0 \) to \( p-1 \)

for \( j = 0 \) to \( p-1 \) except \( i \)

if (\text{myproc} == i) send \( A(i) \) to processor \( j \)

if (\text{myproc} == j) receive \( A(i) \) from processor \( i \)

\( C(\text{myproc}) = C(\text{myproc}) + A(i) \times B(i,\text{myproc}) \)

barrier

Cost of inner loop:

- computation: \( 2 \times n^2 \times (n/p)^2 = 2 \times n^3 / p^2 \)
- communication: \( \alpha + \beta \times n^2 / p \)
Naïve MatMul (continued)

Cost of inner loop:
- computation: \(2^n \cdot (n/p)^2 = 2^n n^2/p^2\)
- 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
\(\Rightarrow\) the algorithm is almost entirely serial

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

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 \(A(\text{myproc})\) into Tmp
  - \(C(\text{myproc}) = C(\text{myproc}) + \text{Tmp} \cdot B(\text{myproc, myproc})\)
  - \(\text{for } j = 1\text{ to } p-1\)
    - Send Tmp to processor \(\text{myproc+1} \mod p\)
    - Receive Tmp from processor \(\text{myproc-1} \mod p\)
  - \(C(\text{myproc}) = C(\text{myproc}) + \text{Tmp} \cdot B(\text{myproc-j mod p, myproc})\)

- Same idea as for gravity in simple sharks and fish algorithm
  - May want double buffering in practice for overlap
  - Ignoring deadlock details in code
  - \(\text{Time of inner loop } = 2^3(\alpha + \beta n^2/p) + 2n^*(n/p)^2\)

- (Nearly) Optimal for 1D layout on Ring or Bus, even with Broadcast:
  - Perfect speedup for arithmetic
  - \(A(\text{myproc})\) must move to each other processor, costs at least \((p-1)\text{cost of sending } n^*(n/p)\) words

- Parallel Efficiency = \(2n^3 / (p \cdot \text{Total Time})\)
  \[= 1/(1 + \alpha \cdot p^2(2n^2) + \beta \cdot p(2n))\]
  \[= 1/(1 + O(pn))\]

- 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
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
**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](http://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](http://www.netlib.org/lapack/lawns/lawn100.ps)

---

**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:
  \[
  C = A \cdot B
  = \sum_k A(:,k) \cdot B(k,:)
  = \sum_k (k-th \text{ col of } A) \cdot (k-th \text{ row of } B)
  \]

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

---

**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 \( B \)
- \( C[i,j] = C[i,j] + \sum_k A[i,k] \cdot B[k,j] \)
  - 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 its value to the whole processor row.
- For all $j = 1$ to $P^{1/2}$
  - Owner of $B[k,j]$ broadcasts its value to the whole processor column.

Receive $A[i,k]$ into $Acol$
Receive $B[k,j]$ into $Brow$

$C_{myproc} = C_{myproc} + Acol \times Brow$

SUMMA Costs

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

- For all $i = 1$ to $P^{1/2}$
  - Owner of $A[i,k]$ broadcasts its value to the whole processor row.
- For all $j = 1$ to $P^{1/2}$
  - Owner of $B[k,j]$ broadcasts its value to the whole processor column.

- $\#\text{words} = \log P \times n^{3/2}$
- $\#\text{messages} = \log P^{1/2}$

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 $c \geq 1$ copies, so $M = cn^2/P$?
  - $\#\text{words}_\text{moved} = \Omega(\#\text{flops} / M^{1/2}) = \Omega( n^2 / (c^{1/2} P^{1/2}) )$
  - $\#\text{messages} = \Omega( \#\text{flops} / M^{3/2} ) = \Omega( P^{1/2} / c^{3/2} )$
- Can we attain new lower bound?
  - Special case: "3D Matmul": $c = P^{1/3}$
    - Bernstein 89, Agarwall, 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 $C(i,j) = C(i,j) + A[i,k] \times B(k,j)$, where each submatrix is $n/P^{1/3} \times n/P^{1/3}$
    - Not always 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: $P = 32$, $c = 2$

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 $\sum A(i,m) \times B(m,j)$
(3) Sum-reduce partial sums $\sum A(i,m) \times B(m,j)$ 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 $\Rightarrow c$ increases proportionally to $P$
- #flops, #words_moved, #messages per proc all decrease proportionally to $P$
- #words_moved = $\Omega(\text{#flops} / M^{1/2}) = \Omega( n^2 / (c^{1/2} P^{1/2}))$
- #messages = $\Omega(\text{#flops} / M^{3/2}) = \Omega( P^{1/2} / c^{3/2})$
- Perfect strong scaling! But only up to $c = P^{1/3}$

Matrix multiplication on 16,384 nodes of BG/P

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

Matrix multiplication on 16,384 nodes of BG/P

- Using $c=16$ matrix copies
- 12X faster
- 2.7X faster
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:
  \[ T(cP) = \frac{n^3}{cP} \left( \gamma_T + \beta_T M^{1/2} + \alpha_T (mM^{1/2}) \right) \]
  with
  \[ \gamma_T, \beta_T, \alpha_T = \text{secs per flop, per word\_moved, per message of size } m \]
- Notation for energy model:
  \[ E(cP) = P \left( \frac{n^3}{cP} \left( \gamma_E + \beta_E M^{1/2} + \alpha_E (mM^{1/2}) \right) + \delta_E MT(cP) + \epsilon_E T(cP) \right) \]
- \( c \) 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

Parallel Strassen

- Complexity of classical Matmul vs Strassen
- Flops: \( O(n^3/p) \) vs \( O(n^w/p) \) where \( w = \log_2 7 \sim 2.81 \)
- Communication lower bound on #words:
  \[ \Omega((n^3/p)/M^{1/2}) = \Omega((n/M^{1/2})^3/p) \]
- Communication lower bound on #messages:
  \[ \Omega((n^3/p)/M^{3/2}) = \Omega((n/M^{1/2})^3/p) \]
- All attainable as M increases, up to a limit:
  can increase M by factor up to \( p^{1/3} \) vs \( p^{1-2/w} \)
  #words as low as \( \Omega(n/p^{2/3}) \) vs \( \Omega(n/p^{2/w}) \)
- Best Paper Prize, SPAA'11, Ballard, D., Holtz, Schwartz

- How well does parallel Strassen work in practice?
Extensions of Lower Bound and Optimal Algorithms

• For each processor that does G flops with fast memory of size M
  \[ \text{#words\_moved} = \Omega \left( \frac{G}{M^{1/2}} \right) \]

• Extension: for any program that “smells like”
  • Nested loops …
  • That access arrays …
  • Where array subscripts are linear functions of loop indices
    • Ex: \( A(i,j), B(3*i-4*k+5*j), i-j, 2*k, \ldots, \ldots \)
  • There is a constant \( s \) such that
    \[ \text{#words\_moved} = \Omega \left( \frac{G}{M^{s-1}} \right) \]
  • \( s \) 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 \( C = A \cdot B \) (1/5)

• 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 \( n \times n \) matrices \( C = A \cdot B \) 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 \( F \cdot \text{#segments} \geq T = \text{total flops} = 2n^3 \), so \( \text{#segments} \geq T / F \)
  • So \( \text{# loads & stores} = M \cdot \text{#segments} \geq M \cdot T / F \)
Proof of Communication Lower Bound on \( C = A \cdot B \) (2/5)

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

Proof of Communication Lower Bound on \( C = A \cdot B \) (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 \( n \times n \times n \) cube
  - Each 1 x 1 subsquare represents one \( A(i,k) \)
  - Each 1 x 1 subsquare represents one \( B(k,j) \)
  - Each 1 x 1 subsquare represents one \( C(i,j) \)
  - Each 1 x 1 x 1 subcube represents one \( C(i,j) += A(i,k) \cdot B(k,j) \)
    - May be added directly to \( C(i,j) \), or to temporary accumulator

Proof of Communication Lower Bound on \( C = A \cdot B \) (4/5)

- 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 \( \leq (2M \cdot 2M \cdot 2M)^{3/2} \approx F \)
- # Segments \( \geq \lfloor 2n^3 / F \rfloor \)
- # Loads & Stores = M \cdot \#Segments \( \geq M \cdot \lfloor 2n^3 / F \rfloor \)
  \( \geq n^3 / (2M)^{3/2} - M = \Omega(n^3 / M^{1/2}) \)
- 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)
  - \( \# \text{ "Load & Stores"} = \# \text{ words moved from or to other procs} \geq M \cdot (2n^3 / P) / F \cdot M \cdot (2n^3 / P) / (2M)^{3/2} = n^2 / (2P)^{1/2} \)