Parallel Numerical Algorithms

- Lecture schedule:
  - 3/8: Dense Matrix Products
    - BLAS 1: Vector operations
    - BLAS 2: Matrix-Vector operations
    - BLAS 3: Matrix-Matrix operations
    - Use of Performance models in algorithm design
  - 3/10: Dense Matrix Solvers
  - 3/12: Dense Matrix Solvers
    - Use of Performance models in algorithm design
  - 3/15: Sparse Matrix Products
  - 3/17: Sparse Direct Solvers

Parallel Vector Operations

Some common vector operations for vectors x,y,z:

- Vector add: z = x + y
  - Trivial to parallelize if vectors are aligned
- AXPY: z = a*x + y (where a is scalar)
  - Broadcast a, followed by independent * and +
- Dot product: s = \sum x[j] * y[j]
  - Independent * followed by + reduction

Broadcast and reduction

- Broadcast of 1 value to p processors in log p time
- Reduction of p values to 1 in log p time
- Takes advantage of associativity in +,*, min, max, etc.

Broadcast Algorithms

- Sequential or "centralized" algorithm
  - P0 sends value to P-1 other processors in sequence
  - O(P) algorithm
  - Note: variations in UPC/Titanium model based on whether P0 writes to all others, or others read from P0
- Tree-based algorithm
  - May vary branching factor
  - O(log P) algorithm
- If broadcasting large data blocks, may break into pieces and pipeline

Lower Bound on Parallel Performance

- To compute a function of n inputs \( x_1, \ldots, x_n \)
  - Given only binary operations on our machine.
    - In 1 time step, output depends on at most 2 inputs
    - In 2 time steps, output depends on at most 4 inputs
    - Adding a time step increases possible inputs by at most 2x
  - In k-log n time steps, output depends on at most n inputs
  - A function of n inputs requires at least log n parallel steps.
Scan (or Parallel prefix), A Digression

- What if you want to compute partial sums
- Definition: the parallel prefix operation take a binary associative operator \( \oplus \), and an array of \( n \) elements
  \[ [a_0, a_1, a_2, \ldots, a_{n-1}] \]
  and produces the array
  \[ [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \ldots \oplus a_{n-1})] \]
- Example: add scan of
  \[ [1, 2, 0, 4, 2, 1, 1, 3] \]
  is \[ [1, 3, 3, 7, 9, 10, 11, 14] \]
- Can be implemented in \( O(n) \) time by a serial algorithm
  - Obvious \( n-1 \) applications of operator will work

Applications of scans

- There are several applications of scans, some more obvious than others
  - lexically compare string of characters
  - add multi-precision numbers (represented as array of numbers)
  - evaluate polynomials
  - implement bucket sort and radix sort
  - solve tridiagonal systems
  - to dynamically allocate processors
  - to search for regular expression (e.g., grep)

Prefix Sum in parallel


- Parallel prefix works on any associative operator
- Updating “odds”
- Names: \(+\) (APL), \(\text{cumsum}(\text{Matlab})\), MPI\_SCAN
- Warning: \( 2n \) operations used when only \( n-1 \) needed

Implementing Scans

- Tree summation 2 phases
  - up sweep
    - get values L and R from left and right child
    - save L in local variable \( \text{Mine} \)
    - compute \( \text{Tmp} = L + R \) and pass to parent
  - down sweep
    - get value \( \text{Tmp} \) from parent
    - send \( \text{Tmp} \) to left child
    - send \( \text{Tmp} + \text{Mine} \) to right child

E.g., Using Scans for Array Compression

- Given an array of \( n \) elements
  \[ [a_0, a_1, a_2, \ldots, a_{n-1}] \]
  and an array of flags
  \[ [1, 0, 1, 0, 0, \ldots] \]
  compress the flagged elements
  \[ [a_0, a_2, a_3, a_6, \ldots] \]
- Compute a “prescan” i.e., a scan that doesn’t include the element at position \( i \) in the sum
  \[ [0, 1, 1, 2, 3, 3, 4, \ldots] \]
- Gives the index of the \( i \)-th element in the compressed array
  - If the flag for this element is 1, write it into the result array at the given position
E.g., Fibonacci via Matrix Multiply Prefix

\[
\begin{bmatrix}
F_{n+1} \\
F_n
\end{bmatrix} =
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
F_n \\
F_{n-1}
\end{bmatrix}
\]

Can compute all \( F_n \) by matmul_prefix on
\[
\begin{bmatrix}
F_0 & F_1 & F_2 & \ldots & F_{n-1} \\
1 & 0 & 0 & \ldots & 0
\end{bmatrix}
\]
then select the upper left entry.

Segmented Operations

Inputs = Ordered Pairs
(operand, boolean)
e.g. \((x, T)\) or \((x, F)\)

<table>
<thead>
<tr>
<th>+</th>
<th>y, T</th>
<th>y, F</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, T)</td>
<td>x+y, T</td>
<td>(y, F)</td>
</tr>
<tr>
<td>(x, F)</td>
<td>(y, T)</td>
<td>(x@y, F)</td>
</tr>
</tbody>
</table>

\[\text{e.g.}\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
T & T & F & F & F & T & F & T
\end{array}
\]
Result:

\[
\begin{array}{cccccc}
1 & 3 & 3 & 7 & 12 & 6 & 7 & 8
\end{array}
\]

End of Digression

Summary of data parallel operations
- Vector add, etc. is embarrassingly parallel
- Broadcast used for axpy operations
- Reduction used for dot product
- Parallel prefix (scan) is a variation on reduction with partial results
  - Useful in parallelizing surprising algorithms
  - If something seems serial, try this

Now back to our regular programming
We have covered the idea with most BLAS1 (vector) operations
Now onto vector/matrix (BLAS2) and matrix-matrix (BLAS3)

Parallel Matrix-Vector Product

- Compute \( y = y + A^T x \), where \( A \) is a dense matrix
  - Layout: 1D by rows
  - Algorithm:
    - Foreach processor \( i \)
    - Broadcast \( x(i) \)
    - Compute \( y(i) = A(i)^T x \)
  - \( A(i) \) 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 uses the formula
    \[
    y(i) = y(i) + A(i)^T x = y(i) + \Sigma \left(A(i)^T x(i)\right)
    \]

Matrix-Vector Product

- A column layout of the matrix eliminates the broadcast
  - But adds a reduction to update the destination
- A 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
  - Topology of machine
  - Scheduling communication
- Use of performance models for algorithm design
  - Message Time = "latency" + #words * time-per-word
  - $\alpha + n^b$

Latency Bandwidth Model

- Network of fixed number $P$ of processors
  - fully connected
  - each with local memory
- Latency ($\alpha$)
  - accounts for varying performance with number of messages
  - gap ($g$) in logP model may be more accurate cost if messages are pipelined
- Inverse bandwidth ($\beta$)
  - accounts for performance varying with volume of data
- Efficiency (in any model):
  - serial time / ($P \times$ parallel time)
  - perfect (linear) speedup $\Rightarrow$ efficiency = 1

Matrix Multiply with 1D Column Layout

- Assume matrices are $n \times n$ and $n$ is divisible by $p$
- $A(i)$ refers to the $n \times n/p$ block column that processor $i$ owns (similarly for $B(i)$ and $C(i)$)
- $B(i,j)$ is the $n/p \times n/p$ sublock of $B(i)$
  - in rows $j \times n/p$ through $(j+1) \times n/p$
- Algorithm uses the formula
  - $C(i) = C(i) + A \times B(i) = C(i) + \sum_j A(j) \times B(i,j)$

Matrix Multiply: 1D Layout on Bus or Ring

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

MatMul: 1D layout on Bus without Broadcast

Naïve algorithm:

```plaintext
C(myproc) = C(myproc) + A(myproc) \times 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) \times B(i,myproc)
barrier
```

Cost of inner loop:

- computation: $2n^2(n/p)^2 = 2n^3/p^2$
- communication: $\alpha + \beta n^2 / p$

Naïve MatMul (continued)

Cost of inner loop:

- computation: $2n^2(n/p)^2 = 2n^3/p^2$
- communication: $\alpha + \beta n^2 / p$

Only 1 pair of processors $i$ and $j$ are active on any iteration, and of those, only $i$ is doing computation $\Rightarrow$ the algorithm is almost entirely serial

Running time:

- $= \left( p(p-1) + 1 \right)^2$ computation + $p^2(p-1)$ communication
- $= 2n^2 + p^2\alpha + p^3n^b$

this is worse than the serial time and grows with $p$
Matmul for 1D layout on a Processor Ring

- Pairs of 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*(α + β*n^2/p) + 2*n*(n/p)^2

Matmul for 1D layout on a Processor Ring

- Time of inner loop = 2*(α + β*n^2/p) + 2*n*(n/p)^2
- Total Time = 2*n'*(n/p)^2 + (p-1)*Time of inner loop
- Optimal for 1D layout on Ring or Bus, even with 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 + α*p^2/(2*n^3) + β*p/(2*n))
    = 1/(1 + O(p/n))
  - Grows to 1 as n/p increases (or α and β shrink)

MatMul with 2D Layout

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

- Assume p is square s x s grid

Cannon’s Algorithm

\[ C(i,j) = C(i,j) + \sum A(i,k) * B(k,j) \]

- Assume s = sqrt(p) is an integer
- forall i=0 to s-1  \hspace{1cm} “skew” A
  left-circular-shift row i of A by i
  so that A(i,j) overwritten by A(i,(j+i)mod s)
  \hspace{1cm} “skew” B
  up-circular-shift B column i of B by i
  so that B(i,j) overwritten by B((i+j)mod s), j)

- for all processors in parallel
  C(i,j) = C(i,j) + A(i,j)*B(i,j)
  left-circular-shift each row of A by 1
  up-circular-shift each row of B by 1

Cannon’s Matrix Multiplication

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

Initial Step to Skew Matrices in Cannon

- Initial blocked input

- After skewing before initial block multiplies

Skewing Steps in Cannon

- First step
  - A(0,0)  A(0,1)  A(0,2)
  - A(1,0)  A(1,1)  A(1,2)
  - A(2,0)  A(2,1)  A(2,2)

- Second
  - A(0,1)  A(1,0)  A(2,0)
  - A(1,1)  A(1,2)  A(2,1)
  - A(2,2)  A(0,2)  A(0,1)

- Third
  - A(0,0)  A(0,2)  A(0,1)
  - A(1,0)  A(1,1)  A(1,2)
  - A(2,0)  A(2,1)  A(2,2)

Cost of Cannon’s Algorithm

forall i=0 to s-1          recall s = sqrt(p)
left-circular-shift row i of A by i  ... cost = s(α + β* n2/p)
forall i=0 to s-1
up-circular-shift B column i of B by i  ... cost = s(α + β* n2/p)
for k=0 to s-1
forall i=0 to s-1 and j=0 to s-1
C(i,j) = C(i,j) + A(i,j)*B(i,j)  ... cost = 2*(n/s) 3 = 2*n3/p
left-circular-shift each row of A by 1  ... cost = α + β* n2/p
up-circular-shift each row of B by 1  ... cost = α + β* n2/p

Total Time = 2*n3/p + 4*s*α + 4*β* n2/s
Parallel Efficiency = 2*n3 / (p * Total Time)
= 1/(1 + α* 2*(s/n)3 + β* 2*(s/n))
= 1/(1 + O(sqrt(p)/n))
Grows to 1 as n/s = n/sqrt(p) = sqrt(data per processor) grows
Better than 1D layout, which had Efficiency = 1/(1 + O(p/n))

Drawbacks to Cannon

- Hard to generalize for
  - p not a perfect square
  - A and B not square
  - Dimensions of A, B not perfectly divisible by s=sqrt(p)
  - A and B not “aligned” in the way they are stored on processors
  - block-cyclic layouts
  - Memory hog (extra copies of local matrices)

SUMMA Algorithm

- SUMMA = Scalable Universal Matrix Multiply
- Slightly less efficient, but simpler and easier to generalize
- 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

- I, J represent all rows, columns owned by a processor
- k is a single row or column
  - or a block of b rows or columns
- C(I,J) = C(I,J) + Σk A(I,k)*B(k,J)
- Assume a pr by pc processor grid (pr = pc = 4 above)
- Need not be square

For k=0 to n-1  ... or b-1 where b is the block size
  ... = # cols in A(I,k) and # rows in B(k,J)
  ... in parallel
for all I = 1 to pr
  owner of A(I,k) broadcasts it to whole processor row
for all J = 1 to pc
  owner of B(k,J) broadcasts it to whole processor column
Receive A(I,k) into Acol
Receive B(k,J) into Brow
C(I,J) = C(I,J) + Σk Acol * Brow
**SUMMA performance**

To simplify analysis only, assume $s = \sqrt{p}$

For $k = 0$ to $n/b - 1$
  for all $i = 1$ to $s$  \[ \ldots \]
  owner of $A(i,k)$ broadcasts it to whole processor row
  \[ \ldots \text{time} = \log s \cdot (\alpha + \beta \cdot b/n/s), \text{using a tree} \]
  for all $j = 1$ to $s$
  owner of $B(k,j)$ broadcasts it to whole processor column
  \[ \ldots \text{time} = \log s \cdot (\alpha + \beta \cdot b/n/s), \text{using a tree} \]
Receive $A(i,k)$ into $A_{col}$
Receive $B(k,j)$ into $B_{row}$
$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + A_{col} \cdot B_{row}$
\[ \ldots \text{time} = 2 \cdot (n/s)^2 \cdot b \]

Total time $= 2 \cdot n^3/p + \alpha \cdot \log p \cdot n/b + \beta \cdot \log p \cdot n^2/s$

---

**SUMMA performance**

- Total time $= 2 \cdot n^3/p + \alpha \cdot \log p \cdot n/b + \beta \cdot \log p \cdot n^2/s$
- Parallel Efficiency $= 1/(1 + \alpha \cdot \log p \cdot n/(2 \cdot \sqrt{n} \cdot b) + \beta \cdot \log p \cdot s/(2 \cdot n))$
- Same $\beta$ term as Cannon, except for $\log p$ factor
  \[ \log p \text{ grows slowly so this is ok} \]
- Latency ($\alpha$) term can be larger, depending on $b$
  When $b=1$, get $\alpha \cdot \log p \cdot n$
  As $b$ grows to $n/s$, term shrinks to $\alpha \cdot \log p \cdot s$ ($\log p$ times Cannon)
- Temporary storage grows like $2b/n/s$
- Can change $b$ to tradeoff latency cost with memory

---

**ScaLAPACK Parallel Library**

For both cache hierarchies and parallelism, recursive layouts may be useful
- Z-Morton, U-Morton, and X-Morton Layout
- Also Hilbert layout and others
- What about the user’s view?
  - Fortunately, many problems can be solved on a permutation
  - Never need to actually change the user’s layout

**Recursive Layouts**

- For both cache hierarchies and parallelism, recursive layouts may be useful
- Z-Morton, U-Morton, and X-Morton Layout

---

**Summary of Parallel Matrix Multiplication**

- **1D Layout**
  - Bus without broadcast - slower than serial
  - Nearest neighbor communication on a ring (or bus with broadcast): Efficiency $= 1/(1 + O(p/n))$
- **2D Layout**
  - Cannon
    - Efficiency $= 1/(1+O(\sqrt{p}/n))$
    - Hard to generalize for general $p, n, \text{block cyclic, alignment}$
  - SUMMA
    - Efficiency $= 1/(1 + O(\log p \cdot n / (b \cdot n^2) + \log p \cdot \sqrt{p}/n))$
    - Very General
    - $b$ small $\gg$ less memory, lower efficiency
    - $b$ large $\gg$ more memory, high efficiency
- **Recursive layouts**
  - Current area of research
Extra Slides

Gaussian Elimination

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

LU Algorithm:
1: Split matrix into two rectangles (m x n/2)
   if only 1 column, scale by reciprocal of pivot & return
2: Apply LU Algorithm to the left part
3: Apply transformations to right part
   (triangular solve $A_{ij} = L^{-1}A_{ij}$ and
   matrix multiplication $A_{ij} = A_{ij} - A_{ij}^*A_{ij}^*$)
4: Apply LU Algorithm to right part

Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, ...

Recursive Factorizations

• Just as accurate as conventional method
• Same number of operations
• Automatic variable blocking
  • Level 1 and 3 BLAS only!
• Extreme clarity and simplicity of expression
• Highly efficient
• The recursive formulation is just a rearrangement of the point-wise
  LINPACK algorithm
• The standard error analysis applies (assuming the matrix
  operations are computed the “conventional” way).

Review: BLAS 3 (Blocked) GEPP

for $ib \leq b \leq n$ step $b$
   ... Process block $b$ of columns at a time
   ... Point to end of block of $b$ columns
   apply BLAS2 version of GEPP to get $A_{ib} = P^* L^* U$
   ... let $LL$ denote the strict lower triangular part of $A_{ib}\ldots A_{ib}$
   ... $A_{ib} = L^{-1}L^T A_{ib}$
   ... apply delayed updates with single matrix-multiply
   ... with inner dimension $b$
   ... apply delayed updates with single matrix-multiply
   ... with inner dimension $b$
Review: Row and Column Block Cyclic Layout

processors and matrix blocks are distributed in a 2d array
pcol-fold parallelism
in any column, and calls to the BLAS2 and BLAS3 on matrices of size brow-by-bcol
serial bottleneck is eased
need not be symmetric in rows and columns

Distributed GE with a 2D Block Cyclic Layout

block size b in the algorithm and the block sizes brow and bcol in the layout satisfy b=brow=bcol.
shaded regions indicate busy processors or communication performed.
unnecessary to have a barrier between each step of the algorithm, e.g. step 9, 10, and 11 can be pipelined

Distributed Gaussian Elimination with a 2D Block Cyclic Layout

\[ \text{for } k = 1 \rightarrow n-1 \text{ step b} \]
\[ \text{for } i = 1 \rightarrow \text{brow} \text{ step b} \]
\[ m(i, j) = m(i, j) - A(i, j) \times m(j, k) \]

Matrix multiply of green = green - blue * pink

LAPACK and ScalAPACK

Machines: Workstations, Vector, SMP
Distributed: Processors, DSM
Based on BLAS: BLAS, BLACS
Functionality: Linear Systems, Least Squares, Eigenproblems
Linear Systems, Least Squares, Eigenproblems (less than LAPACK)
Matrix types: Dense, banded, out-of-core
Error Bounds: Complete
Languages: Fortran and C
Interface to: C, Fortran, HPF
Manual? Yes
### Scales well, nearly full machine speed

Old version, pre 1998 Gordon Bell Prize Still have ideas to accelerate Project Available!

### Old Algorithm, plan to abandon

Have good ideas to speedup Project available!

Hardest of all to parallelize Have alternative, and would like to compare Project available!

### Out-of-core means matrix lives on disk; too big for main mem
Much harder to hide latency of disk

### QR much easier than LU because no pivoting needed for QR
Moral: use QR to solve Ax=b Projects available (perhaps very hard...)

---

### Work-Depth Model of Parallelism

- The work depth model:
  - The simplest model is used
  - For algorithm design, independent of a machine
  - The work, W, is the total number of operations
  - The depth, D, is the longest chain of dependencies
  - The parallelism, P, is defined as W/D

- Specific examples include:
  - circuit model, each input defines a graph with ops at nodes
  - vector model, each step is an operation on a vector of elements
  - language model, where set of operations defined by language