CS 267
Parallel Matrix Multiplication

Kathy Yelick

http://www.cs.berkeley.edu/~yelick/cs267
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: Matrix Multiply context results (HW1)
    • 310 Soda at 1:30pm
  • 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: \( z = a \times x + y \) (where \( a \) is scalar)
  - Broadcast \( a \), followed by independent * and +
- Dot product: \( s = \sum_j x[j] \times y[j] \)
  - 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

$\Rightarrow$ 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 $\ominus$, and an array of $n$ elements $[a_0, a_1, a_2, \ldots a_{n-1}]$ and produces the array $[a_0, (a_0 \ominus a_1), \ldots (a_0 \ominus a_1 \ominus \ldots \ominus a_{n-1})]$.

- 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


slide source: Alan Edelman, MIT
Parallel Prefix Cost

- Parallel prefix works on any associative operator

  1  2  3  4  5  6  7  8
  \|  \|  \|  \|  \|  \|  \|  \|  Pairwise sums
  3  7  11  15
  |  |  |  |  |  |  |  Recursive prefix
  3  10  21  36
  \|  \|  \|  \|  \|  Update “odds”
  1  3  6  10  15  21  28  36

- Names: +\ (APL), cumsum(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 Mine
    - compute $\text{Tmp} = L + R$ and pass to parent
  - **Down sweep**
    - get value Tmp from parent
    - send Tmp to left child
    - send Tmp+Mine to right child

Up sweep:
- $\text{mine} = \text{left}$
- $\text{tmp} = \text{left} + \text{right}$

Down sweep:
- $\text{tmp} = \text{parent (root is 0)}$
- $\text{right} = \text{tmp} + \text{mine}$
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
  \[ 1,0,1,1,0,0,1,\ldots \]
  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

\[ F_{n+1} = F_n + F_{n-1} \]

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

Can compute all \( F_n \) by matmul\_prefix on

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\]

then select the upper left entry
## Segmented Operations

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

<table>
<thead>
<tr>
<th>Change of segment indicated by switching T/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 2</td>
</tr>
<tr>
<td>(x, T)</td>
</tr>
<tr>
<td>(x+y, T)</td>
</tr>
<tr>
<td>(y, T)</td>
</tr>
<tr>
<td>(x⊕y, F)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e. g.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>Result</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
The “Myth” of log $n$

• The $\log_2 n$ parallel steps is **not** the main reason for the usefulness of parallel prefix.

• Say $n = 1000p$ (1000 summands per processor)
  • Cost = $(2000 \text{ adds}) + (\log_2 P \text{ message passings})$

  fast & embarassingly parallel

  (2000 local adds are serial for each processor of course)
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 \( y = y + A \times x \), where \( A \) is a dense matrix

• Layout:
  • 1D by rows

• Algorithm:
  • Foreach processor \( i \)
  • Broadcast \( x(i) \)
  • Compute \( y(i) = A(i) \times x \)

• \( A(i) \) 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
  \[
  y(i) = y(i) + A(i) \times x = y(i) + \sum_j A(i) \times x(j)
  \]
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: $2n^3$ Flops
- Variables are:
  - Data layout
  - Topology of machine
  - Scheduling communication

- Use of performance models for algorithm design
  - Message Time = “latency” + \#words * time-per-word
    \[ = \alpha + n \beta \]
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 \( n \) by \( n/p \) block column that processor \( i \) owns (similarly for \( B(i) \) and \( C(i) \))

- \( B(i,j) \) is the \( n/p \) by \( n/p \) sublock of \( B(i) \)
  - in rows \( j*n/p \) through \( (j+1)*n/p \)

- 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
  \[ C(i) = C(i) + A*B(i) = C(i) + \sum_j A(j)*B(j,i) \]

• 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:

\[ C(\text{myproc}) = C(\text{myproc}) + A(\text{myproc}) \times B(\text{myproc}, \text{myproc}) \]
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: \( 2 \times n \times (n/p)^2 = 2 \times n^3/p^2 \)
- communication: \( \alpha + \beta \times n^2 \) / \( p \)
Naïve MatMul (continued)

Cost of inner loop:
  computation: $2n^3 / p^2$
  communication: $\alpha + \beta n^2 / p \quad \ldots \text{approximately}$

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:
  $= (p(p-1) + 1) \text{computation} + \ p(p-1) \text{communication}$
  $\sim 2n^3 + \ p^2 \alpha + \ p^2 n^2 \beta$

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

- Pairs of processors can communicate simultaneously

\[
\begin{align*}
\text{Copy } A(\text{myproc}) & \text{ into } Tmp \\
C(\text{myproc}) & = C(\text{myproc}) + Tmp \cdot B(\text{myproc}, \text{myproc}) \\
\text{for } j = 1 \text{ to } p-1 & \\
& \quad \text{Send } Tmp \text{ to processor } \text{myproc}+1 \mod p \\
& \quad \text{Receive } Tmp \text{ from processor } \text{myproc}-1 \mod p \\
C(\text{myproc}) & = C(\text{myproc}) + Tmp \cdot B(\text{myproc}-j \mod p, \text{myproc})
\end{align*}
\]

- 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 = \(2(\alpha + \beta \cdot n^2/p) + 2n \cdot (n/p)^2\)
Matmul for 1D layout on a Processor Ring

- Time of inner loop = $2(a + b n^2/p) + 2n(n/p)^2$
- Total Time = $2n(n/p)^2 + (p-1) \times $ Time of inner loop
- $\sim 2n^3/p + 2p a + 2b n^2$

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

- Parallel Efficiency = $2n^3 / (p \times \text{Total Time})$
  = $1/(1 + \alpha * p^2/(2n^3) + \beta * p/(2n))$
  = $1/(1 + O(p/n))$

- Grows to 1 as $n/p$ increases (or $\alpha$ and $\beta$ 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

```
\begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array}
```

• Assume p is square s x s grid
Cannon’s Algorithm

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

... assume \( s = \sqrt{p} \) is an integer

forall \( i=0 \) to \( s-1 \) \( \text{ “skew” A} \)
left-circular-shift row \( i \) of \( A \) by \( i \)
... so that \( A(i,j) \) overwritten by \( A(i,(j+i) \mod s) \)
forall \( i=0 \) to \( s-1 \) \( \text{ “skew” B} \)
up-circular-shift column \( i \) of \( B \) by \( i \)
... so that \( B(i,j) \) overwritten by \( B((i+j) \mod s), j \)
for \( k=0 \) to \( s-1 \) \( \text{ sequential} \)
forall \( i=0 \) to \( s-1 \) and \( j=0 \) to \( s-1 \) \( \text{ all processors in parallel} \)
\[ C(i,j) = C(i,j) + A(i,j)B(i,j) \]
left-circular-shift each row of \( A \) by 1
up-circular-shift each row of \( B \) by 1
Cannon’s Matrix Multiplication

Cannon's Matrix Multiplication Algorithm

\[
C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2)
\]
### Initial Step to Skew Matrices in Cannon

- **Initial blocked input**

<table>
<thead>
<tr>
<th>A(0,0)</th>
<th>A(0,1)</th>
<th>A(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,0)</td>
<td>A(1,1)</td>
<td>A(1,2)</td>
</tr>
<tr>
<td>A(2,0)</td>
<td>A(2,1)</td>
<td>A(2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(0,0)</th>
<th>B(0,1)</th>
<th>B(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,0)</td>
<td>B(1,1)</td>
<td>B(1,2)</td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(2,1)</td>
<td>B(2,2)</td>
</tr>
</tbody>
</table>

- **After skewing before initial block multiplies**

<table>
<thead>
<tr>
<th>A(0,0)</th>
<th>A(0,1)</th>
<th>A(0,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(1,1)</td>
<td>A(1,2)</td>
<td>A(1,0)</td>
</tr>
<tr>
<td>A(2,2)</td>
<td>A(2,0)</td>
<td>A(2,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(0,0)</th>
<th>B(1,1)</th>
<th>B(2,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
</tr>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
</tr>
</tbody>
</table>
Skewing Steps in Cannon

- First step

- Second

- Third
**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/p³/2
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\alpha + 4*\beta*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

- \( C(I,J) = C(I,J) + \sum_k A(I,k) * B(k,J) \)

- Assume a \( p_r \) by \( p_c \) processor grid (\( p_r = p_c = 4 \) above)
  - Need not be square
SUMMA

For $k=0$ to $n-1$ ... or $n/b-1$ where $b$ is the block size

... = # cols in $A(I,k)$ and # rows in $B(k,J)$

for all $I = 1$ to $p_r$ ... in parallel

owner of $A(I,k)$ broadcasts it to whole processor row

for all $J = 1$ to $p_c$ ... in parallel

owner of $B(k,J)$ broadcasts it to whole processor column

Receive $A(I,k)$ into $A_{col}$

Receive $B(k,J)$ into $B_{row}$

$$C(\text{myproc}, \text{myproc}) = C(\text{myproc}, \text{myproc}) + A_{col} \times B_{row}$$
SUMMA performance

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

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

° Total time $= 2 \times n^3/p + \alpha \times \log p \times n/b + \beta \times \log p \times n^2/s$
SUMMA performance

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

ScaLAPACK SOFTWARE HIERARCHY

- ScaLAPACK
- PBLAS
- LAPACK
- BLACS
- BLAS
- Message Passing Primitives (MPI, PVM, etc.)

Global

Local
**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
Recursive Layouts

- 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
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, block cyclic, alignment
  • SUMMA
    • Efficiency = \(1/(1 + O(\log p \cdot p / (b \cdot n^2) + \log p \cdot \sqrt{p}/n))\)
    • Very General
    • \(b\) small => less memory, lower efficiency
    • \(b\) large => more memory, high efficiency

• Recursive layouts
  • Current area of research
Extra Slides
Gaussian Elimination

**Standard Way**
subtract a multiple of a row

**LINPACK**
apply sequence to a column

**LAPACK**
apply sequence to nb
then apply nb to rest of matrix

\[ a_2 = L^{-1} a_2 \]
\[ a_3 = a_3 - a_1 a_2 \]
Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

LU Algorithm:
1: Split matrix into two rectangles \((m \times n/2)\)
   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 \(A_{12} = L^{-1}A_{12}\) and
    matrix multiplication \(A_{22} = A_{22} - A_{21} A_{12}\))

4: Apply LU Algorithm to right part

Most of the work in the matrix multiply
Matrices of size \(n/2, n/4, n/8, \ldots\)

Slide source: Dongarra
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).
Pentium III 550 MHz Dual Processor
LU Factorization

Dual-processor
LAPACK
Recursive LU

Uniprocessor
LAPACK
Recursive LU

Slide source: Dongarra
Review: BLAS 3 (Blocked) GEPP

\[ \text{for } \text{ib} = 1 \text{ to } n-1 \text{ step } b \quad \text{... Process matrix } b \text{ columns at a time} \]
\[ \text{end} = \text{ib} + b-1 \quad \text{... Point to end of block of } b \text{ columns} \]
\[ \text{apply BLAS2 version of GEPP to get } A(\text{ib:}n, \text{ib:end}) = P^\prime \ast L^\prime \ast U^\prime \]
\[ \text{... let } LL \text{ denote the strict lower triangular part of } A(\text{ib:end}, \text{ib:end}) + I \]
\[ A(\text{ib:end}, \text{end+1:n}) = LL^{-1} \ast A(\text{ib:end}, \text{end+1:n}) \quad \text{... update next } b \text{ rows of } U \]
\[ A(\text{end+1:n}, \text{end+1:n}) = A(\text{end+1:n}, \text{end+1:n}) - A(\text{end+1:n}, \text{ib:end}) \ast A(\text{ib:end}, \text{end+1:n}) \quad \text{... apply delayed updates with single matrix-multiply} \]
\[ \text{... with inner dimension } b \]

Gaussian Elimination using BLAS 3

[Diagram showing the process of Gaussian elimination using BLAS 3]
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 $b_{row}$ and $b_{col}$ in the layout satisfy $b=b_{row}=b_{col}$.

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

\[ \text{for } \text{ib} = 1 \text{ to } n-1 \text{ step b} \]
\[ \text{end} = \min(\text{ib+b-1, n}) \]

\[ \text{for } i = \text{ib} \text{ to } \text{end} \]

1. find pivot row \( k \), column broadcast

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

3. \( A(\text{i+1:n, i}) = A(\text{i+1:n, i}) / A(\text{i, i}) \)

4. \( A(\text{i+1:n, i+1:end}) = A(\text{i+1:n, i}) * A(\text{i, i+1:end}) \)

end for

5. broadcast all swap information right and left

6. apply all rows swaps to other columns
(7) Broadcast $LL_{right}$

(8) $A(\text{ib:end, end+1:n}) = LL \setminus A(\text{ib:end, end+1:n})$

(9) Broadcast $A(\text{ib:end, end+1:n})_{down}$

(10) Broadcast $A(\text{end+1:n, ib:end})_{right}$

(11) Eliminate $A(\text{end+1:n, end+1:n})$
PDGESV = ScaLAPACK parallel LU routine

Since it can run no faster than its inner loop (PDGEMM), we measure:

Efficiency = Speed(PDGESV)/Speed(PDGEMM)

Observations:
- Efficiency well above 50% for large enough problems
- For fixed N, as P increases, efficiency decreases (just as for PDGEMM)
- For fixed P, as N increases, efficiency increases (just as for PDGEMM)
- From bottom table, cost of solving Ax=b about half of matrix multiply for large enough matrices.
- From the flop counts we would expect it to be \( \frac{2n^3}{(2/3)n^3} = 3 \) times faster, but communication makes it a little slower.
# LAPACK and ScaLAPACK

<table>
<thead>
<tr>
<th></th>
<th>LAPACK</th>
<th>ScaLAPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machines</strong></td>
<td>Workstations, Vector, SMP</td>
<td>Distributed Memory, DSM</td>
</tr>
<tr>
<td><strong>Based on</strong></td>
<td>BLAS</td>
<td>BLAS, BLACS</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Linear Systems</td>
<td>Linear Systems</td>
</tr>
<tr>
<td></td>
<td>Least Squares</td>
<td>Least Squares</td>
</tr>
<tr>
<td></td>
<td>Eigenproblems</td>
<td>Eigenproblems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(less than LAPACK)</td>
</tr>
<tr>
<td><strong>Matrix types</strong></td>
<td>Dense, band</td>
<td>Dense, band,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>out-of-core</td>
</tr>
<tr>
<td><strong>Error Bounds</strong></td>
<td>Complete</td>
<td>A few</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>F77 or C</td>
<td>F77 and C</td>
</tr>
<tr>
<td><strong>Interfaces to</strong></td>
<td>C++, F90</td>
<td>HPF</td>
</tr>
<tr>
<td><strong>Manual?</strong></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Where?</strong></td>
<td><a href="http://www.netlib.org/lapack">www.netlib.org/lapack</a></td>
<td><a href="http://www.netlib.org/scalapack">www.netlib.org/scalapack</a></td>
</tr>
</tbody>
</table>
Performance of ScaLAPACK QR (Least squares)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>Efficiency</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>4000</td>
<td>10000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>.54</td>
<td>.61</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.46</td>
<td>.55</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.26</td>
<td>.60</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.51</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.29</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.19</td>
<td>.54</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>4</td>
<td>32</td>
<td>.61</td>
<td>.36</td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td></td>
<td>.43</td>
<td>.62</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.22</td>
<td>.62</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.51</td>
<td>.77</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.49</td>
<td>.66</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.37</td>
<td>.71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>Time</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>4000</td>
<td>10000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>1.2</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.5</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.6</td>
<td>1.1</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>1.3</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>2.3</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>3.6</td>
<td>1.2</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>4</td>
<td>32</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td></td>
<td>1.6</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>3.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>1.3</td>
<td>.9</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>1.4</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.8</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Scales well, nearly full machine speed
Old version, pre 1998 Gordon Bell Prize
Still have ideas to accelerate Project Available!

### Performance of Symmetric Eigensolvers

**Time(PDSYEVX)/Time(PDGBM)**

(bisection + inverse iteration)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray T3E</td>
<td>4</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>16</td>
<td>50</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

**Time(PDSYEV)/Time(PDGBM)**

(QR iteration)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray T3E</td>
<td>4</td>
<td>32</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>16</td>
<td>50</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>193</td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>
Have good ideas to speedup Project available!

Performance of SVD (Singular Value Decomposition)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
<th>2000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>67</td>
<td>66</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td>93</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>97</td>
<td>60</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>72</td>
<td>38</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance of Nonsymmetric Eigensolver (QR iteration)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
<th>1000</th>
<th>1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel XP/S MP</td>
<td>16</td>
<td>50</td>
<td>123</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td>Paragon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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...)

3/9/2004
A small software project ...

Participants

<table>
<thead>
<tr>
<th>Krste Asanovic (UC Berkeley)</th>
<th>Zhaojun Bai (U Kentucky)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richard Barrett (U Tenn)</td>
<td>Michael Berry (U Tenn)</td>
</tr>
<tr>
<td>Jeff Bilmes (UC Berkeley)</td>
<td>Chris Bischof (ANL)</td>
</tr>
<tr>
<td>Susan Blackford (ORNL)</td>
<td>Soumen Chakrabarti (UC Berkeley)</td>
</tr>
<tr>
<td>Tony Chan (UCLA)</td>
<td>Chee-Whye Chin (UC Berkeley)</td>
</tr>
<tr>
<td>Jaeyoung Choi (LBNL)</td>
<td>Andy Cleary (LLNL)</td>
</tr>
<tr>
<td>Ed D'Asveza (ORNL)</td>
<td>Jim Demmel (UC Berkeley)</td>
</tr>
<tr>
<td>Inderjit Dhillon (UC Berkeley)</td>
<td>June Donato (ORNL)</td>
</tr>
<tr>
<td>Jack Dongarra (U Tenn, ORNL)</td>
<td>Zlatko Drmač (U Hagen)</td>
</tr>
<tr>
<td>Jeremy Du Croz (NAG)</td>
<td>Victor Eijkhout (UCLA)</td>
</tr>
<tr>
<td>Stan Eisenstat (Yale)</td>
<td>Vince Fernando (NAG)</td>
</tr>
<tr>
<td>John Gilbert (Xerox PARC)</td>
<td>Ming Gu (UC Berkeley, LBL)</td>
</tr>
<tr>
<td>Sven Hammarling (NAG)</td>
<td>Mike Heath (U Illinois)</td>
</tr>
<tr>
<td>Greg Henry (Intel)</td>
<td>Dominic Lam (UC Berkeley)</td>
</tr>
<tr>
<td>Steve Huss-Lederman (SRC)</td>
<td>Bo Kågström (U Umeå)</td>
</tr>
<tr>
<td>W. Kahan (UC Berkeley)</td>
<td>Youngdae Kim (U Tenn)</td>
</tr>
<tr>
<td>RenCang Li (UC Berkeley)</td>
<td>Xiaoye Li (UC Berkeley)</td>
</tr>
<tr>
<td>Joseph Liu (York)</td>
<td>Beresford Parlett (UC Berkeley)</td>
</tr>
<tr>
<td>Antoine Petitet (U Tenn)</td>
<td>Peter Poromaa (U Umeå)</td>
</tr>
<tr>
<td>Roland Pozo (U Tenn)</td>
<td>Padma Raghavan (U Illinois)</td>
</tr>
<tr>
<td>Huan Ren (UC Berkeley)</td>
<td>Howard Robinson (UC Berkeley)</td>
</tr>
<tr>
<td>Charles Romaine (ORNL)</td>
<td>Jeff Rutter (UC Berkeley)</td>
</tr>
<tr>
<td>Ivan Slapničar (U Split)</td>
<td>Dan Sorensen (U Rice)</td>
</tr>
<tr>
<td>Ken Stanley (UC Berkeley)</td>
<td>Xiaobai Sun (ANL)</td>
</tr>
<tr>
<td>Bernard Tourancheau (U Tenn)</td>
<td>Anna Tsao (SRC)</td>
</tr>
<tr>
<td>Robert van de Geijn (U Texas)</td>
<td>Henk van der Vorst (Utrecht U)</td>
</tr>
<tr>
<td>Paul Van Dooren (U Illinois)</td>
<td>Krešimir Veselić (U Hagen)</td>
</tr>
<tr>
<td>David Walker (ORNL)</td>
<td>Clint Whaley (U Tenn)</td>
</tr>
</tbody>
</table>

With the cooperation of
Cray, IBM, Convex, DEC, Fujitsu, NEC, NAG, IMSL

3/9/2004

Supported by ARPA, NSF, DOE
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