CS 267
Dense Linear Algebra: Parallel Matrix Multiplication

James Demmel

www.cs.berkeley.edu/~dемmel/cs267_Spr07
Outline

• Recall BLAS = Basic Linear Algebra Subroutines
• Matrix-vector multiplication in parallel
• Matrix-matrix multiplication in parallel
# Review of the BLAS

- Building blocks for all linear algebra
- Parallel versions call serial versions on each processor
  - So they must be fast!
- Recall \( q = \frac{\text{# flops}}{\text{# mem refs}} \)
  - The larger is \( q \), the faster the algorithm can go in the presence of memory hierarchy
- “axpy”:\( y = \alpha x + y \), where \( \alpha \) scalar, \( x \) and \( y \) vectors

<table>
<thead>
<tr>
<th>BLAS level</th>
<th>Ex.</th>
<th># mem refs</th>
<th># flops</th>
<th>( q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“Axpy”, Dot prod</td>
<td>3n</td>
<td>2n(^1)</td>
<td>2/3</td>
</tr>
<tr>
<td>2</td>
<td>Matrix-vector mult</td>
<td>( n^2 )</td>
<td>2n(^2)</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Matrix-matrix mult</td>
<td>4n(^2)</td>
<td>2n(^3)</td>
<td>( n/2 )</td>
</tr>
</tbody>
</table>
Different Parallel Data Layouts for Matrices

Why? Want parallelism within submatrices

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 $y = y + A \cdot x$, 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 $y(i) = A(i) \cdot x$

• Algorithm uses the formula
  $$y(i) = y(i) + A(i) \cdot x = y(i) + \sum_j A(i,j) \cdot x(j)$$
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 (of $x(j)$ in a processor column) and reduction (of $A(i,j)^*x(j)$ in a processor row),
  - $\sqrt{p}$ by $\sqrt{p}$ for square processor grid
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 (of \( x(j) \) in a processor column) and reduction (of \( A(i,j)*x(j) \) in a processor row),
  - \( \sqrt{p} \) by \( \sqrt{p} \) for square processor grid

It may be useful to have \( x(i) \) and \( y(i) \) on same proc
Parallel Matrix Multiply

- Computing C = C + A*B
- Using basic algorithm: 2*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
    \[ = \alpha + n \beta \]
  - \( \alpha \) and \( \beta \) measured in #flop times (i.e. time per flop = 1)
- Efficiency (in any model):
  - serial time / (p * parallel time)
  - perfect (linear) speedup \( \leftrightarrow \) efficiency = 1
Matrix Multiply with 1D Column Layout

• Assume matrices are n x n and n is divisible by p

![Diagram showing column layout with processors p0, p1, p2, ..., p7]

• 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
  \[ C(i) = C(i) + A*B(i) = C(i) + \sum_j A(j)*B(j,i) \]
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 each block column \( A(i) \) of \( A \)
for \( j = 0 \) to \( p-1 \) except \( i \) … for every processor not having \( A(i) \)
\[\begin{align*}
\text{if (myproc == i)} & \text{ send } A(i) \text{ to processor } j \\
\text{if (myproc == j)} & \text{ receive } A(i) \text{ from processor } i \\
C(\text{myproc}) & = C(\text{myproc}) + A(i) \times B(i, \text{myproc}) \\
\text{barrier}
\end{align*}\]

Cost of inner loop:

\[
\begin{align*}
\text{computation: } & 2 \times n \times (n/p)^2 = 2 \times n^3/p^2 \\
\text{communication: } & \alpha + \beta \times n^2 / p
\end{align*}
\]
Naïve MatMul (continued)

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

Only 1 pair of processors (i and j) are active on any iteration, and of those, only i is doing computation
=> the algorithm is almost entirely serial

Running time:
= $(p^*(p-1) + 1)^*\text{computation} + p^*(p-1)^*\text{communication}$
~ $2n^3 + p^2\alpha + pn^2\beta$

This is worse than the serial time and grows with p.

Why might you still want to do this?
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*(\(\alpha + \beta n^2/p\)) + 2*n*(n/p)^2
- Total Time = 2*n* (n/p)^2 + (p-1) * Time of inner loop
- \(~ 2*n^3/p + 2*p*\alpha + 2*\beta*n^2~
- 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 = \(2*n^3 / (p * \text{Total Time})\)
  = 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)
MatMul with 2D Layout

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

\[ \begin{array}{c|c|c} p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \end{array} \]

\[ = \]

\[ \begin{array}{c|c|c} p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \end{array} \]

\[ \times \]

\[ \begin{array}{c|c|c} p(0,0) & p(0,1) & p(0,2) \\ \hline p(1,0) & p(1,1) & p(1,2) \\ \hline p(2,0) & p(2,1) & p(2,2) \\ \end{array} \]

- Assume \( p \) processors form square \( s \times s \) 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 $s-1$               ... “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 $s-1$               ... “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 $s-1$            ... sequential
forall $i=0$ to $s-1$ and $j=0$ to $s-1$   ... 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 column of $B$ by 1
Cannon’s Matrix Multiplication

Cannon’s Matrix Multiplication Algorithm

<table>
<thead>
<tr>
<th>Initial A, B</th>
<th>A, B after skewing</th>
<th>A, B after shift k=1</th>
<th>A, B after shift k=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(0,0) A(0,1) A(0,2)</td>
<td>A(0,0) A(0,1) A(0,2)</td>
<td>A(0,0) A(0,2) A(0,0)</td>
<td>A(0,2) A(0,2) A(0,1)</td>
</tr>
<tr>
<td>A(1,0) A(1,1) A(1,2)</td>
<td>A(1,1) A(1,2) A(1,0)</td>
<td>A(0,0) A(0,2) A(1,1)</td>
<td>A(1,1) A(1,2) A(1,0)</td>
</tr>
<tr>
<td>A(2,0) A(2,1) A(2,2)</td>
<td>A(2,2) A(2,2) A(2,1)</td>
<td>A(0,0) A(0,2) A(1,1)</td>
<td>A(1,1) A(1,2) A(1,0)</td>
</tr>
</tbody>
</table>

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

- Initial blocked input

- After skewing
Shifting Steps in Cannon

• First step

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

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

• Second

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

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

• Third

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,0)</td>
<td>B(0,1)</td>
<td>B(1,2)</td>
</tr>
<tr>
<td>B(0,0)</td>
<td>B(1,1)</td>
<td>B(2,2)</td>
</tr>
<tr>
<td>B(1,0)</td>
<td>B(2,1)</td>
<td>B(0,2)</td>
</tr>
</tbody>
</table>
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*n^2/p)
forall i=0 to s-1
  up-circular-shift column i of B by i … cost = s*(α + β*n^2/p)
for k=0 to s-1        …sequential loop
  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*n^3/p^3/2
  left-circular-shift each row of A by 1  … cost = α + β*n^2/p
  up-circular-shift each column of B by 1  … cost = α + β*n^2/p

° Total Time = 2*n^3/p + 4*s*α + 4*β*n^2/s
° Parallel Efficiency = 2*n^3 / (p * Total Time)
  = 1/( 1 + s/n)°* 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))
Pros and Cons of Cannon

• Local computation one call to (optimized) matrix-multiply

• 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](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

- 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

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

- Assume a \( p_r \) by \( p_c \) processor grid (\( p_r = p_c = 4 \) above)
  - Need not be square
For $k=0$ to $n-1$ ... or $n/b-1$ where $b$ is the block size

... $\equiv$ # 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 $B_{row}$

$C_{_myproc} = C_{_myproc} + A_{col} \ast B_{row}$
SUMMA performance

° To simplify analysis only, assume s = sqrt(p)

For k=0 to n/b-1
  for all i = 1 to s   ...  s = sqrt(p)
    owner of A(i,k) broadcasts it to whole processor row
    ... time = log s *( α + β * b*n/s), using a tree
  for all j = 1 to s
    owner of B(k,j) broadcasts it to whole processor column
    ... time = log s *( α + β * b*n/s), using a tree
Receive A(i,k) into Acol
Receive B(k,j) into Brow
C_myproc = C_myproc + Acol * Brow
... time = 2*(n/s)^2*b

° Total time = 2*n^3/p  +  α * log p * n/b  +  β * log p * n^2 /s
SUMMA performance

- Total time = \(2n^3/p + \alpha \log p n/b + \beta \log p n^2/s\)
- Parallel Efficiency =
  \[
  \frac{1}{1 + \alpha \log p \frac{p}{(2b)n^2} + \beta \log p \frac{s}{2n}}
  \]
- ~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 \(\alpha \log p n\)
  As \(b\) grows to \(n/s\), term shrinks to
  \(\alpha \log p s\) (\(\log p\) times Cannon)
- Temporary storage grows like \(2b n/s\)
- 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

02/21/2007

---

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4=2x2</td>
<td>32</td>
<td>1055</td>
</tr>
<tr>
<td></td>
<td>16=4x4</td>
<td></td>
<td>3630</td>
</tr>
<tr>
<td></td>
<td>64=8x8</td>
<td></td>
<td>13456</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>755</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>2514</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>6205</td>
</tr>
<tr>
<td>Intel XP/S MP</td>
<td>4</td>
<td>32</td>
<td>330</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1233</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>4496</td>
</tr>
<tr>
<td>Paragon</td>
<td>4</td>
<td>32</td>
<td>463</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>2490</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>4130</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>470</td>
</tr>
<tr>
<td></td>
<td>32=4x8</td>
<td></td>
<td>2822</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>5457</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Machine</th>
<th>Peak/ proc</th>
<th>DGEMM MFlops</th>
<th>Procs</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>600</td>
<td>360</td>
<td>4</td>
<td>.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>.58</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>266</td>
<td>200</td>
<td>4</td>
<td>.94</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>.48</td>
</tr>
<tr>
<td>Intel XP/S MP</td>
<td>100</td>
<td>90</td>
<td>4</td>
<td>.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>.78</td>
</tr>
<tr>
<td>Paragon</td>
<td>334</td>
<td>129</td>
<td>4</td>
<td>.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>.50</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>334</td>
<td>129</td>
<td>4</td>
<td>.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>.50</td>
</tr>
</tbody>
</table>
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?
  - Some problems can be solved on a permutation
  - May not need to actually change the user’s layout
  - Or: convert on input/output, invisibly to user
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 = $1/(1+O(\alpha \cdot (\sqrt{p}/n)^3 + \beta \cdot \sqrt{p}/n))$
    • Hard to generalize for general $p$, $n$, block cyclic, alignment
  • SUMMA
    • Efficiency = $1/(1 + O(\alpha \cdot \log p \cdot p / (b \cdot n^2) + \beta \cdot \log p \cdot \sqrt{p}/n))$
    • 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

Slide source: Dongarra
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).
DGEMM ATLAS & DGETRF Recursive

AMD Athlon 1GHz (~$1100 system)

Pentium III 550 MHz Dual Processor
LU Factorization

Recursive LU

Dual-processor

Uniprocessor

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

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

Gaussian Elimination using BLAS 3

---

02/21/2007
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

\[ \text{for } ib = 1 \text{ to } n-1 \text{ step } b \]

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

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

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

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

3. \[ A( i+1:n, i ) = A( i+1:n, i ) / A(i, i) \]

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

\[ \text{end for} \]

5. Broadcast all swap information right and left

6. Apply all rows swaps to other columns
Matrix multiply of green = green - blue * pink

(7) Broadcast LL right

(8) \[ A(ib:end, end+1:n) = LL \setminus A(ib:end, end+1:n) \]

(9) Broadcast \( A(ib:end, end+1:n) \) down

(10) Broadcast \( A(end+1:n, ib:end) \) right

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

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

Efficiency = \[
\frac{\text{Speed}(\text{PDGESV})}{\text{Speed}(\text{PDGEMM})}
\]

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 \( \frac{2n^3}{2/3n^3} = 3 \) times faster, but communication makes it a little slower.

### Performance of ScaLAPACK LU

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>N</th>
<th>2000</th>
<th>4000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>.67</td>
<td>.82</td>
<td>.84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.44</td>
<td>.65</td>
<td>.75</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.18</td>
<td>.47</td>
<td>.75</td>
<td></td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.56</td>
<td>.52</td>
<td>.66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.29</td>
<td>.52</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.15</td>
<td>.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel XP/S MP</td>
<td>4</td>
<td>32</td>
<td>.64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td></td>
<td>.37</td>
<td>.66</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.16</td>
<td>.42</td>
<td>.75</td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.76</td>
<td>.62</td>
<td>.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.28</td>
<td>.54</td>
<td>.69</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>N</th>
<th>2000</th>
<th>4000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>.50</td>
<td>.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.75</td>
<td>.51</td>
<td>.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.86</td>
<td>.72</td>
<td>.45</td>
<td></td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.16</td>
<td>.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.24</td>
<td>1.03</td>
<td>.51</td>
<td></td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>4</td>
<td>32</td>
<td>.52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td></td>
<td>.89</td>
<td>.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.08</td>
<td>.79</td>
<td>.44</td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.44</td>
<td>.54</td>
<td>.47</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.88</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.18</td>
<td>.62</td>
<td>.49</td>
<td></td>
</tr>
</tbody>
</table>
# 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, 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/">www.netlib.org/</a> scalapack](<a href="http://www.netlib.org/">http://www.netlib.org/</a> scalapack)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Machine</th>
<th>Poles</th>
<th>Block N</th>
<th>N</th>
<th>2000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray T3E</td>
<td>4</td>
<td>32</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM SP2</td>
<td>16</td>
<td>50</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>64</td>
<td></td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Poles</th>
<th>Block N</th>
<th>N</th>
<th>2000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gray T3E</td>
<td>4</td>
<td>32</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>37</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM SP2</td>
<td>16</td>
<td>50</td>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>64</td>
<td></td>
<td>193</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Have good ideas to speedup Project available!

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N 2000</th>
<th>N 4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>66</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>93</td>
<td>70</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>38</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>59</td>
<td>26</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 1000</th>
<th>N 1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel XP/S MP</td>
<td>16</td>
<td>50</td>
<td>123</td>
<td>97</td>
</tr>
<tr>
<td>Paragon</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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...)

Out-of-Core Performance Results for Least Squares

- Prototype code for Out-of-Core extension
- Linear solvers based on “Left-looking” variants of LU, QR, and Cholesky factorization
- Portable I/O interface for reading/writing ScaLA-PACK matrices

QR Factorization on 64 processors Intel Paragon

[Bar chart showing comparison between Out-of-core and In-core performance for various problem sizes.]
A small software project ...

Participants

Krstie Asanovic (UC Berkeley)  Zhaojun Bai (U Kentucky)
Richard Barrett (U Tenn)  Michael Berry (U Tenn)
Jeff Bilmes (UC Berkeley)  Chris Bischof (ANL)
Susan Blackford (ORNL)  Soumen Chakrabarti (UC Berkeley)
Tony Chan (UCLA)  Chee-Whye Chin (UC Berkeley)
Jaeyoung Choi (LLNL)  Andy Cleary (LLNL)
Ed D'Azevedo (ORNL)  Jim Demmel (UC Berkeley)
Indrjit Dhillon (UC Berkeley)  June Donato (ORNL)
Jack Dongarra (U Tenn, ORNL)  Zlatko Drmač (U Hagen)
Jeremy Du Croz (NAG)  Victor Eijkhout (UCLA)
Stan Eisenstat (Yale)  Vince Fernando (NAG)
John Gilbert (Xerox PARC)  Ming Gu (UC Berkeley, LBL)
Sven Hammarling (NAG)  Mike Heath (U Illinois)
Greg Henry (Intel)  Dominic Lam (UC Berkeley)
Steve Huss-Lederman (SRC)  Bo Kågström (U Umeå)
W. Kahan (UC Berkeley)  Youngae Kim (U Tenn)
Hao Li (UC Berkeley)  Xiaoye Li (UC Berkeley)
Joseph Liu (York)  Beresford Parlett (UC Berkeley)
Antoine Petitet (U Tenn)  Peter Papamarcou (U Umeå)
Roldan Pozo (U Tenn)  Padmini Raghavan (U Illinois)
Huan Ren (UC Berkeley)  Howard Robinson (UC Berkeley)
Charles Romine (ORNL)  Jeff Rutter (UC Berkeley)
Ivan Slapnicar (U Split)  Dan Sorensen (Hrice U)
Ken Stanley (UC Berkeley)  Xiaowei Sun (ANL)
Bernard Tourancheau (U Tenn)  Anna Tsao (SRC)
Robert van de Geijn (U Texas)  Henk van der Vorst (Utrecht U)
Paul Van Dooren (U Illinois)  Kreimir Veselić (U Hagen)
David Walker (ORNL)  Clint Whaley (U Tenn)
Kathy Yelick (UC Berkeley)

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

02/21/2007

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
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 / ($p$ * parallel time)
  - perfect (linear) speedup $\rightarrow$ efficiency = 1
Initial Step to Skew Matrices in Cannon

• Initial blocked input

\[
\begin{array}{ccc}
A(0,0) & A(0,1) & A(0,2) \\
A(1,0) & A(1,1) & A(1,2) \\
A(2,0) & A(2,1) & A(2,2) \\
\end{array}
\quad
\begin{array}{ccc}
B(0,0) & B(0,1) & B(0,2) \\
B(1,0) & B(1,1) & B(1,2) \\
B(2,0) & B(2,1) & B(2,2) \\
\end{array}
\]

• After skewing before initial block multiplies

\[
\begin{array}{ccc}
A(0,0) & A(0,1) & A(0,2) \\
A(1,1) & A(1,2) & A(1,0) \\
A(2,2) & A(2,0) & A(2,1) \\
\end{array}
\quad
\begin{array}{ccc}
B(0,0) & B(1,1) & B(2,2) \\
B(1,0) & B(2,1) & B(0,2) \\
B(2,0) & B(0,1) & B(1,2) \\
\end{array}
\]
Skewing Steps in Cannon

- **First step**

  - A(0,0) | A(0,1) | A(0,2)  
  - A(1,1) | A(1,2) | A(1,0)  
  - A(2,2) | A(2,0) | A(2,1)  
  - B(0,0) | B(1,1) | B(2,2)  
  - B(1,0) | B(2,1) | B(0,2)  
  - B(2,0) | B(0,1) | B(1,2)  

- **Second**

  - A(0,1) | A(0,2) | A(0,0)  
  - A(1,2) | A(1,0) | A(1,1)  
  - A(2,0) | A(2,1) | A(2,2)  
  - B(1,0) | B(2,1) | B(0,2)  
  - B(2,0) | B(0,1) | B(1,2)  

- **Third**

  - A(0,2) | A(0,0) | A(0,1)  
  - A(1,0) | A(1,1) | A(1,2)  
  - A(2,1) | A(2,2) | A(2,0)  
  - B(2,0) | B(0,1) | B(1,2)  
  - B(0,0) | B(1,1) | B(2,2)  
  - B(1,0) | B(2,1) | B(0,2)