ScalAPACK SOFTWARE HIERARCHY

ScalAPACK

PBLAS

Local

Global

LAPACK

BLACS

Message Passing Primitives (MPI, PVM, etc.)

BLAS
# LAPACK and ScaLAPACK

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

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>N 2000</th>
<th>N 4000</th>
<th>N 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>.54</td>
<td>.61</td>
<td>.60</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.46</td>
<td>.55</td>
<td>.60</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.26</td>
<td>.47</td>
<td>.54</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.51</td>
<td>.51</td>
<td>.54</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.29</td>
<td>.54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.19</td>
<td>.36</td>
<td>.54</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>4</td>
<td>32</td>
<td>.61</td>
<td>.63</td>
<td>.62</td>
</tr>
<tr>
<td>Paragon</td>
<td>16</td>
<td></td>
<td>.43</td>
<td>.48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.22</td>
<td>.48</td>
<td>.54</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.51</td>
<td>.77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.49</td>
<td>.66</td>
<td>.71</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.37</td>
<td>.60</td>
<td>.72</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

### Performance of Symmetric Eigensolvers

<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>10</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>13</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>29</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>16</td>
<td>50</td>
<td>24</td>
<td>40</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>22</td>
<td>34</td>
<td>20</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>24</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Performance of Symmetric Eigensolvers (QR iteration)

<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>35</td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>37</td>
<td></td>
<td>37</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>58</td>
<td>47</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>35</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The “Holy Grail” of Eigensolvers for Symmetric matrices

Symmetric tridiagonal eigenproblem on ALPHA EV6: times relative to DSTEGR

To be propagated throughout LAPACK and ScaLAPACK
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>Time(PDGESVD)/Time(PDGEMM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>67</td>
<td>66 64</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>66</td>
<td>66 64</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>93</td>
<td>93 70</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>97</td>
<td>60 81</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>60</td>
<td>60 81</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>81</td>
<td>81 70</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>72</td>
<td>38 16</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>38</td>
<td>38 16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>59</td>
<td>59 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</th>
<th>Time(PDLAHQR)/Time(PDGEMM)</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>123</td>
<td>97</td>
</tr>
</tbody>
</table>

Hardest of all to parallelize
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

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](chart.jpg)
A small software project ...

Participants

Krste 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)  Saumen 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)
Inderjit 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)  Youngdae Kim (U Tenn)
Rencang Li (UC Berkeley)  Xiaoye Li (UC Berkeley)
Joseph Liu (York)  Beresford Parlett (UC Berkeley)
Antoine Petitet (U Tenn)  Peter Poromaa (U Umeå)
Roldan Pozo (U Tenn)  Padma Raghavan (U Illinois)
Huan Ren (UC Berkeley)  Howard Robinson (UC Berkeley)
Charles Romine (ORNL)  Jeff Rutter (UC Berkeley)
Ivan Slapničar (U Split)  Dan Sorensen (Rice U)
Ken Stanley (UC Berkeley)  Xiaobai 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)  Kresimir 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

Supported by ARPA, NSF, DOE
Extra Slides
Parallelizing Gaussian Elimination

° Recall parallelization steps from Lecture 3
   • **Decomposition**: identify enough parallel work, but not too much
   • **Assignment**: load balance work among threads
   • **Orchestrate**: communication and synchronization
   • **Mapping**: which processors execute which threads

° Decomposition
   • In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with \( n^2 \) processors, need \( 3n \) parallel steps

   ```
   for i = 1 to n-1
   A(i+1:n,i) = A(i+1:n,i) / A(i,i)         ... BLAS 1 (scale a vector)
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n )      ... BLAS 2 (rank-1 update)
   - A(i+1:n , i) * A(i , i+1:n)
   ```

   • This is too fine-grained, prefer calls to local matmuls instead
Assignment of parallel work in GE

° Think of assigning submatrices to threads, where each thread responsible for updating submatrix it owns
  • “owner computes” rule natural because of locality

° What should submatrices look like to achieve load balance?
Different Data Layouts for Parallel GE (on 4 procs)

Bad load balance:
P0 idle after \( n/4 \) steps

1) Column Blocked Layout

<table>
<thead>
<tr>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

2) Column Cyclic Layout

<table>
<thead>
<tr>
<th>bcol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
</tr>
<tr>
<td>2 3 2 3</td>
</tr>
<tr>
<td>0 1 0 1</td>
</tr>
<tr>
<td>2 3 2 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>brow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1</td>
</tr>
<tr>
<td>2 3 2 3</td>
</tr>
<tr>
<td>0 1 0 1</td>
</tr>
<tr>
<td>2 3 2 3</td>
</tr>
</tbody>
</table>

Can trade load balance and BLAS2/3 performance by choosing \( b \), but factorization of \( t \) column is a bottleneck

3) Column Block Cyclic Layout

| 0 1 2 3 |
| 1 2 3 0 |
| 2 3 0 1 |
| 3 0 1 2 |

4) Row and Column Block Cyclic Layout

5) Block Skewed Layout
The main steps in the solution process are

Fill: computing the matrix elements of A

Factor: factoring the dense matrix A

Solve: solving for one or more excitations b

Field Calc: computing the fields scattered from the objec
## Analysis of MOM for Parallel Implementation

<table>
<thead>
<tr>
<th>Task</th>
<th>Work</th>
<th>Parallelism</th>
<th>Parallel Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>$O(n^{**2})$</td>
<td>embarrassing</td>
<td>low</td>
</tr>
<tr>
<td>Factor</td>
<td>$O(n^{**3})$</td>
<td>moderately diff.</td>
<td>very high</td>
</tr>
<tr>
<td>Solve</td>
<td>$O(n^{**2})$</td>
<td>moderately diff.</td>
<td>high</td>
</tr>
<tr>
<td>Field Calc.</td>
<td>$O(n)$</td>
<td>embarrassing</td>
<td>high</td>
</tr>
</tbody>
</table>
BLAS 3 (Blocked) GEPP, using Delayed Updates

for \( ib = 1 \) to \( n-1 \) step \( b \)  
\hspace{1cm} \ldots \text{Process matrix } b \text{ columns at a time}

\hspace{1cm} \text{end} = ib + b-1  
\hspace{1cm} \ldots \text{Point to end of block of } b \text{ columns}

\hspace{1cm} \text{apply BLAS2 version of GEPP to } \text{get } A(ib:n, \text{ib:end}) = P' * L' * U'

\hspace{1cm} \ldots \text{let } LL \text{ denote the strict lower triangular part of } A(ib:end, \text{ib:end}) + I

\hspace{1cm} A(ib:end, \text{end+1:n}) = LL^{-1} * A(ib:end, \text{end+1:n})  
\hspace{1cm} \ldots \text{update next } b \text{ rows of } U

\hspace{1cm} A(end+1:n, \text{end+1:n} ) = A(end+1:n, \text{end+1:n} )  
\hspace{1cm} - A(end+1:n, \text{ib:end}) * A(ib:end, \text{end+1:n})  
\hspace{1cm} \ldots \text{apply delayed updates with single matrix-multiply}

\hspace{1cm} \ldots \text{with inner dimension } b

Gaussian Elimination using BLAS 3

\hspace{1cm} \text{Completed part of } U

\hspace{1cm} \text{Completed part of } L

\hspace{1cm} LL

\hspace{1cm} A(ib:end, \text{ib:end})

\hspace{1cm} A(end+1:n, \text{ib:end})

\hspace{1cm} A(ib:end, \text{end+1:n})

\hspace{1cm} A(end+1:n, \text{end+1:n})
for $i = 1$ to $n-1$
    find and record $k$ where $|A(k,i)| = \max\{i \leq j \leq n\} |A(j,i)|$
    ... i.e. largest entry in rest of column $i$
    if $|A(k,i)| = 0$
        exit with a warning that $A$ is singular, or nearly so
    elseif $k \neq i$
        swap rows $i$ and $k$ of $A$
    end if
    $A(i+1:n,i) = A(i+1:n,i) / A(i,i)$
    ... each quotient lies in $[-1,1]$
    ... BLAS 1
    $A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) * A(i, i+1:n)$
    ... BLAS 2, most work in this line
How to proceed:

° Consider basic parallel matrix multiplication algorithms on simple layouts
  • Performance modeling to choose best one
    - Time (message) = latency + #words * time-per-word
    - = α + n*β

° Briefly discuss block-cyclic layout

° PBLAS = Parallel BLAS
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
1D Layout

- Assume matrices are \( n \times n \) and \( n \) is divisible by \( p \)

- \( A(i) \) refers to the \( n \times \frac{n}{p} \) block column that processor \( i \) owns (similarly for \( B(i) \) and \( C(i) \))

- \( B(i,j) \) is the \( \frac{n}{p} \times \frac{n}{p} \) subblock of \( B(i) \)
  - in rows \( j \times \frac{n}{p} \) through \( (j+1) \times \frac{n}{p} \)

- Algorithm uses the formula
  \[
  C(i) = C(i) + A \times B(i) = C(i) + \sum_{j} A(j) \times B(j,i)
  \]
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
Naïve MatMul for 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:** \(2 \cdot n \cdot (n/p)^2 = 2 \cdot n^3/p^2\)
- **communication:** \(\alpha + \beta \cdot n^2/p\) … 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: \((p^*(p-1) + 1) \cdot \text{computation} + p^*(p-1) \cdot \text{communication}\)

\(~= 2 \cdot n^3 + p^2 \cdot \alpha + p \cdot n^2 \cdot \beta\)

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

- Proc i can communicate with Proc(i-1) and Proc(i+1) simultaneously for all i

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

- Same idea as for gravity in simple sharks and fish algorithm

- 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
  ~ 2*n^3/p + 2*p*α + 2*β*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)*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

\[ \begin{array}{ccc}
  p(0,0) & p(0,1) & p(0,2) \\
  p(1,0) & p(1,1) & p(1,2) \\
  p(2,0) & p(2,1) & p(2,2) \\
\end{array} \]
Cannon’s Algorithm

\[ C(i,j) = C(i,j) + \sum_{k} A(i,k) \times B(k,j) \]

... 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 B 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
for \( i = 0 \) to \( s - 1 \) and \( j = 0 \) to \( s - 1 \) ... all processors in parallel
\[ C(i,j) = C(i,j) + A(i,j) \times B(i,j) \]
left-circular-shift each row of A by 1
up-circular-shift each row of B by 1
Communication in Cannon

Cannon's Matrix Multiplication Algorithm

\[
C(1,2) = A(1,0) \times B(0,2) + A(1,1) \times B(1,2) + A(1,2) \times B(2,2)
\]
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 B column i of B by i ... cost = s*(α + β*n^2/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*n^3/p^{3/2}

left-circular-shift each row of A by 1 ... cost = α + β*n^2/p

up-circular-shift each row 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)

° 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 = Scalable Universal Matrix Multiply Algorithm

° 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)

For \( k=0 \) to \( n-1 \) ... or \( n/b-1 \) where \( b \) is the block size
\( \ldots = \# \) 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 \( Acol \)
Receive \( B(k,J) \) into \( Brow \)
\( C(\text{myproc, myproc}) = C(\text{myproc, myproc}) + Acol \times Brow \)
SUMMA performance

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
      ... 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(l,k) into Acol
Receive B(k,J) into Brow
C( myproc , myproc ) = C( myproc , 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
° Parallel Efficiency = 1/(1 + α * log p * p / (2*b*n^2) + β * log p * s/(2*n) )
° ~Same β term as Cannon, except for log p factor
   log p grows slowly so this is ok
° Latency (α) term can be larger, depending on b
   When b=1, get α * log p * n
   As b grows to n/s, term shrinks to α * log p * s (log p times Cannon)
° Temporary storage grows like 2*b*n/s
° Can change b to tradeoff latency cost with memory
Summary of Parallel Matrix Multiply Algorithms

° 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 = $1/(1+O(p^{1/2}/n))$
    - Hard to generalize for general p, n, block cyclic, alignment
  • SUMMA
    - Efficiency = $1/(1 + O(\log p \cdot p / (b \cdot n^2) + \log p \cdot p^{1/2} / n))$
    - Very General
    - b small => less memory, lower efficiency
    - b large => more memory, high efficiency
  • Gustavson et al
    - Efficiency = $1/(1 + O(p^{1/3} / n) )$ ??