Matrix Computations: Direct Methods II

May 5, 2014
Lecture 11
Summary

• You have seen an example of how a typical matrix operation (an important one) can be reduced to using lower level BLAS routines that would have been optimized for a given hardware platform.

• Any other matrix operation that you may think of (matrix-matrix multiply, Cholesky factorization, QR, Householder method, etc) can be constructed from BLAS subprograms in a similar fashion and in fact have been constructed in the package called LAPACK.

• Note that only Level 1 and 2 BLAS routines have been used in LU decomposition. Efficiency considerations?
Overview of LAPACK

• Standard library for dense/banded linear algebra
  – Linear systems: $A\times x = b$
  – Least squares problems: $\min_x \| A\times x - b \|_2$
  – Eigenvalue problems: $Ax = \lambda x$, $Ax = \lambda Bx$
  – Singular value decomposition (SVD): $A = U\Sigma V^T$

• Algorithms reorganized to use BLAS3 as much as possible

• Basis of math libraries on many computers

• Many algorithmic innovations remain
Performance of LAPACK (n=1000)

GESV = LU Decomposition with partial pivoting and row interchanges

GEEV = Eigenvalues and left and right eigenvectors

SYEVD = Eigenvalues and eigenvectors of symmetric matrix

GESVD = SVD of rectangular matrix

GELS = Least-squares solution or minimum norm solution using QR or LQ factorizations
Performance of LAPACK \((n=100)\)

- **GESV**: LU Decomposition with partial pivoting and row interchanges
- **GEEV**: Eigenvalues and left and right eigenvectors
- **SYEVD**: Eigenvalues and eigenvectors of symmetric matrix
- **GESVD**: SVD of rectangular matrix
- **GELS**: Least-squares solution or minimum norm solution using QR or LQ factorizations
Summary, Cont’d

• Need to devise algorithms that can make use of Level 3 BLAS (matrix-matrix) routines, for several reasons:
  – Level 3 routines are known to run much more efficiently due to larger ratio of computation to memory references/communication
  – Parallel algorithms on distributed memory machines will require that we decompose the original matrix into blocks which reside in each processor
  – Parallel algorithms will require that we minimize the surface-to-volume ratio of our decompositions, and blocking becomes the natural approach.
Converting BLAS2 to BLAS3 in GEPP

• Blocking
  – Used to optimize matrix-multiplication
  – Harder here because of data dependencies in GEPP

• Delayed Updates
  – Save updates to “trailing matrix” from several consecutive BLAS2 updates
  – Apply many saved updates simultaneously in one BLAS3 operation

• Same idea works for much of dense linear algebra
  – Open questions remain

• Need to choose a block size $b$ (k in Dongarra’s book)
  – Algorithm will save and apply $b$ updates
  – $b$ must be small enough so that active submatrix consisting of $b$ columns of $A$ fits in cache
  – $b$ must be large enough to make BLAS3 fast
Parallel Algorithms for Dense Matrices

- All that follows is applicable to dense or full matrices only. Square matrices discussed, but arguments are valid for rectangular matrices as well.

- Typical parallelization steps
  - Decomposition: identify parallel work and partitioning
  - Mapping: which processors execute which portion of the work
  - Assignment: load balance work among threads/processors
  - Organization: communication and synchronization
Parallel Algorithms for Dense Matrices

• The thread/processor that owns a given portion of a matrix is responsible for doing all of the computation that involves that portion of the matrix.

• This is the sensible thing to do since communication is minimized (although, due to data dependencies within the matrix, it will still be necessary).

• The question is: how should we subdivide a matrix so that parallel efficiency is maximized? There are various options.
Different Data Layouts for Parallel GE (on 4 procs)

1) Column Blocked Layout

Can trade load balance and BLAS2/3 performance by choosing b, but factorization of block column is a bottleneck.

2) Column Cyclic Layout

Load balanced, but can’t easily use BLAS2 or BLAS3.

The winner!

3) Column Block Cyclic Layout

4) Row and Column Block Cyclic Layout

Complicated addressing

5) Block Skewed Layout
Parallel Matrix Transpose - Block Partition

• Start with simple example: obtain the transpose of a matrix, $A$ defined as:

$$A^T[i, j] = A[j, i], 0 \leq i, j \leq n$$

• All elements below diagonal move above the diagonal and vice versa.

• Assume it takes unit time to exchange a pair of matrix elements. Sequential time of transposing an $n \times n$ matrix is given by

$$\frac{(n^2 - n)}{2}$$

• Consider parallel architectures organized in both a 2D mesh and hypercube structures
Parallel Matrix Transpose - 2D Mesh

- Elements/blocks on lower-left part of matrix move up to the diagonal, and then right to their final location. Each step taken requires communication.

- Elements/blocks on upper-right part of matrix move down to the diagonal, and then left to their final location. Each step taken requires communication.
Parallel Matrix Transpose - 2D Mesh

- If each of the \( p \) processors contains a single number, after all of these communication steps, the matrix has been transposed.
- However, if each processor contains a sub-block of the matrix (the more typical situation), after all blocks have been communicated to their final locations, they need to be locally transposed. Each sub-block will contain \((n/\sqrt{p}) \times (n/\sqrt{p})\) elements and the cost of communication will be higher than before.
- Cost of communication is dominated by elements/blocks that reside in the top-right and bottom-left corners. They have to take an approximate number of hops equal to \(2\sqrt{p}\)
Parallel Matrix Transpose - 2D Mesh

• Each block contains $n^2 / p$ elements, so it takes

$$2(t_s + t_w n^2 / p)\sqrt{p}$$

• for all blocks to move to their final destinations. After that, the local blocks need to be transposed, which can be done in an amount of time equal to

$$n^2 / (2p)$$

• for a total wall clock time equal to

$$T_p = \frac{n^2}{2p} + 2t_s \sqrt{p} + 2t_w \frac{n^2}{\sqrt{p}}$$

• Summing over all $p$ processors, the total time consumed by the parallel algorithm is of order

$$T_{TOTAL} = \Theta(n^2 \sqrt{p})$$

• which is a higher cost than the sequential complexity (order $n^2$). This algorithm, on a 2D mesh is not cost optimal. The same is true regardless of whether store-and-forward or cut-through routing schemes are used
Parallel Matrix Transpose - Hypercube

- This algorithm is called recursive subdivision and maps naturally onto a hypercube.
- After the blocks have all been transposed, the elements inside each block (local to a processor) still need to be transposed.
- Wall clock time
  \[ T_p = \frac{n^2}{2p} + (t_s + t_w \frac{n^2}{\sqrt{p}}) \log p \]
  Total time
  \[ T_{\text{TOTAL}} = \Theta(n^2 \log p) \]
Parallel Matrix Transpose - Hypercube

- With cut-through routing, the timing improves slightly to

\[ T_p = (t_s + t_w \frac{n^2}{p} + 2t_h) \log p \Rightarrow T_{TOTAL} = \Theta(n^2 \log p) \]

- which is still suboptimal

- Using striped partitioning (a.k.a column blocked layout) and cut-through routing on a hypercube, the total time becomes cost-optimal

\[ T_p = \frac{n^2}{2p} + t_s (p - 1) + t_w \frac{n^2}{p} + \frac{1}{2} t_h p \log p \Rightarrow T_{TOTAL} = \Theta(n^2) \]

- Note that this type of partitioning may be cost-optimal for the transpose operation. But not necessarily for other matrix operations, such as LU factorization and matrix-matrix multiply
Blocked Algorithms - LU Factorization

\[ A = LU = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & L_{12} & L_{13} \\ L_{21} & L_{22} & L_{23} \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ U_{22} & U_{23} \\ U_{33} \end{pmatrix} \]

\[ A_{11} = L_{11}U_{11} \quad A_{12} = L_{11}U_{12} \quad A_{13} = L_{11}U_{13} \]

\[ A_{21} = L_{21}U_{11} \quad A_{22} = L_{21}U_{12} + L_{22}U_{22} \quad A_{23} = L_{21}U_{13} + L_{22}U_{23} \]

\[ A_{31} = L_{31}U_{11} \quad A_{32} = L_{31}U_{12} + L_{32}U_{22} \quad A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33} \]
**Blocked Algorithms - LU Factorization**

\[ A_{11} = L_{11} U_{11} \quad A_{12} = L_{11} U_{12} \quad A_{13} = L_{11} U_{13} \]
\[ A_{21} = L_{21} U_{11} \quad A_{22} = L_{21} U_{12} + L_{22} U_{22} \quad A_{23} = L_{21} U_{13} + L_{22} U_{23} \]
\[ A_{31} = L_{31} U_{11} \quad A_{32} = L_{31} U_{12} + L_{32} U_{22} \quad A_{33} = L_{31} U_{13} + L_{32} U_{23} + L_{33} U_{33} \]

- With these relationships, we can develop different algorithms by choosing the order in which operations are performed.
- Blocksize, \( b \), needs to be chosen carefully. \( b=1 \) produces the usual single-processor algorithm. \( b>1 \) will improve performance on a single-processor.
- Three natural variants:
  - Left-Looking LU
  - Right-Looking LU, implemented in LAPACK and ScaLAPACK
  - Crout LU
- Handout on details right-looking algorithm on web page
Blocked Algorithms - LU Factorization

- Assume you have already done the first row and column of the GEPP

\[
A = LU = \begin{pmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{pmatrix} = \begin{pmatrix}
L_{11} & L_{12} & L_{13} \\
L_{21} & L_{22} & L_{23} \\
L_{31} & L_{32} & L_{33}
\end{pmatrix} \begin{pmatrix}
U_{11} & U_{12} & U_{13} \\
U_{21} & U_{22} & U_{23} \\
U_{31} & U_{32} & U_{33}
\end{pmatrix}
\]

- And you have the sub-block below left to work on

\[
\begin{pmatrix}
A_{22} & A_{23} \\
A_{32} & A_{33}
\end{pmatrix}
\]

- Notice that the LU decomposition in this sub-block is independent of the portion you have already completed
Blocked Algorithms - LU Factorization

• For simplicity, change notation of sub-block to

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} = P
\begin{pmatrix}
L_{11} & U_{12} \\
L_{21} & U_{22}
\end{pmatrix} = P
\begin{pmatrix}
L_{11}U_{11} & L_{11}U_{12} \\
L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22}
\end{pmatrix}
$$

• Notice that, once you have done Gaussian Elimination on $A_{11}$ and $A_{21}$ you have already obtained $L_{11}$, $L_{21}$, and $U_{11}$

• Now you can re-arrange the block equations by substituting:

$$
U_{12} \leftarrow (L_{11})^{-1} A_{12}
$$

$$
\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12}
$$

• And repeat the procedure recursively
Blocked Algorithms - LU Factorization

- A graphical view of what is going on is given by:
Blocked Algorithms - LU Factorization

Left-Looking LU  Right-Looking LU  Crout LU

- Variations in algorithm are due to the order in which submatrix operations are performed. Slight advantages to Crout’s algorithm (hybrid of the first two) for vector machines with large memory bandwidth; requires fewer memory references
- Used right-looking algorithm for old homework

Pre-computed sub-blocks
Currently being operated on sub-blocks
Review: BLAS 3 (Blocked) GEPP

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

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

4) Row and Column Block Cyclic Layout
Row and Column Block Cyclic Layout

- In LU factorization, distribution of work becomes uneven as the computation progresses.
- Larger block sizes result in greater load imbalance but reduce frequency of communication between processes.
- Block size controls these tradeoffs.
- Also, some hot spots arise in situations where some processors need to do more work between synchronization points than others (e.g., partial pivoting over rows in a single block-column...other processors stay idle. Also, the computation of each block row of the U factorization requires the solution of a lower triangular system across processes in a single row).
- Processor decomposition controls this type of tradeoff.

(a) block distribution over 2 x 3 grid.
(b) data distribution from processor point-of-view.
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.
- See Dongarra’s book for more details.
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) \( A(i+1:n, i+1:end) = A(i+1:n,i) * A(i,i+1:end) \)

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

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

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

(11) Eliminate $A(\text{end+1:n, end+1:n})$
ScaLAPACK

A Software Library for Linear Algebra Computations on Distributed-Memory Computers
AVAILABLE SOFTWARE:
Dense, Band, and Tridiagonal Linear Systems
- general
- symmetric positive definite
Full-Rank Linear Least Squares
Standard and Generalized
Orthogonal Factorizations
Eigenproblems
- SEP: Symmetric Eigenproblem
- NEP: Nonsymmetric Eigenproblem
- GSEP: Generalized Symmetric Eigenproblem
SVD
Prototype Codes
- HPF interface to ScaLAPACK
- Matrix Sign Function for Eigenproblems
- Out-of-core solvers (LU, Cholesky, QR)
- Super LU
- PBLAS (algorithmic blocking and no
  alignment restrictions.)

DOCUMENTATION:
ScaLAPACK Users' Guide
http://www.netlib.org/scalapack/slug/scalapack_slug.html
Future Work
- Out-of-core Eigensolvers
- Divide and Conquer routines
- C++ and Java Interfaces
Commercial Use
ScaLAPACK has been incorporated into
the following software packages:
- NAG Numerical Library
- IBM Parallel ESSL
- SGI Cray Scientific Software Library
and is being integrated into the VNI IMSL
Numerical Library, as well as software
libraries for Fujitsu, HP/Convex, Hitachi,
and NEC.

http://www.netlib.org/scalapack/
Other Dense Matrix Factorizations

• Details vary slightly, but overall procedure is quite similar
• LU, QR, and Cholesky factorizations can be effectively dealt with within this framework.
• See handout for details if you are interested
• You can also see the details of the ScaLAPACK source to understand the algorithm better.
PDGESV = ScaLAPACK parallel LU routine

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

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

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

### Performance of ScaLAPACK LU

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</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>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.44</td>
<td>.65</td>
<td>.75</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.18</td>
<td>.47</td>
<td>.75</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.56</td>
<td>.52</td>
<td>.66</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.29</td>
<td>.52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.15</td>
<td>.32</td>
<td></td>
</tr>
<tr>
<td>Intel XP/S MP</td>
<td>4</td>
<td>32</td>
<td>.64</td>
<td>.66</td>
<td>.75</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.37</td>
<td>.66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.16</td>
<td>.42</td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>4</td>
<td>32</td>
<td>.76</td>
<td>.62</td>
<td>.71</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.38</td>
<td>.62</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.28</td>
<td>.54</td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.60</td>
<td>.54</td>
<td>.69</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.44</td>
<td>.54</td>
<td>.47</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.18</td>
<td>.62</td>
<td>.49</td>
</tr>
</tbody>
</table>

### Time(PDGESV)/Time(PDGEMM)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</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>.40</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.75</td>
<td>.51</td>
<td>.40</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.86</td>
<td>.72</td>
<td>.45</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.60</td>
<td>.64</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.16</td>
<td>.64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.24</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>4</td>
<td>32</td>
<td>.52</td>
<td>.50</td>
<td>.44</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.89</td>
<td>.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.08</td>
<td>.79</td>
<td></td>
</tr>
<tr>
<td>Paragon</td>
<td>4</td>
<td>32</td>
<td>.44</td>
<td>.54</td>
<td>.47</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.88</td>
<td>.54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.18</td>
<td>.62</td>
<td></td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.44</td>
<td>.54</td>
<td>.47</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.88</td>
<td>.54</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.18</td>
<td>.62</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/scalapack">www.netlib.org/scalapack</a></td>
</tr>
</tbody>
</table>
Scales well, nearly full machine speed

### Performance of ScALAPACK QR (Least squares)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>( \text{Efficiency} = \frac{\text{MFlops}(\text{PDGELS})}{\text{MFlops}(\text{PDGEMM})} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>.54</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.46</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.26</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.29</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.19</td>
</tr>
<tr>
<td>Intel XP/S GP Paragon</td>
<td>4</td>
<td>32</td>
<td>.61</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>.43</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.22</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>.51</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>.49</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>.37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Proc</th>
<th>Block Size</th>
<th>( \frac{\text{Time}(\text{PDGELS})}{\text{Time}(\text{PDGEMM})} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>2.6</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>50</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>3.6</td>
</tr>
<tr>
<td>Intel XP/S GP Paragon</td>
<td>4</td>
<td>32</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>3.0</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>4</td>
<td>32</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>1.8</td>
</tr>
</tbody>
</table>
### Performance of Symmetric Eigensolvers

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>4000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>TBB SP2</td>
<td>16</td>
<td>50</td>
<td>24</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>40</td>
<td>29</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td>Paragon</td>
<td>64</td>
<td></td>
<td>34</td>
<td>20</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>24</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>4000</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4</td>
<td>32</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>37</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>57</td>
<td>41</td>
</tr>
<tr>
<td>TBB SP2</td>
<td>16</td>
<td>50</td>
<td>38</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td>58</td>
<td>47</td>
</tr>
<tr>
<td>Intel XP/S GP</td>
<td>16</td>
<td>32</td>
<td>99</td>
<td>193</td>
</tr>
<tr>
<td>Paragon</td>
<td>64</td>
<td></td>
<td>193</td>
<td>193</td>
</tr>
<tr>
<td>Berkeley NOW</td>
<td>16</td>
<td>32</td>
<td>31</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td>35</td>
<td>55</td>
</tr>
</tbody>
</table>

Old Algorithm, plan to abandon
### Performance of SVD (Singular Value Decomposition)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>N</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2000</td>
<td>4000</td>
<td></td>
</tr>
<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></td>
<td></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></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>Intel XP/S MP Paragon</td>
<td>16</td>
<td>50</td>
<td>123</td>
<td>97</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 $Ax=b$
Other Dense Matrix Algorithms

- ScaLAPACK implements
  - Sparse Gaussian Elimination
  - Direct Sparse Solvers
  - Eigenvalue / eigenvector computation
  - Singular Value Decomposition
  - etc…

- using the same block cyclic partitioning
- Links on web page for more info.