CS 267 Applications of Parallel Computers

Dense Linear Algebra

James Demmel
Outline

° Motivation for Dense Linear Algebra (as opposed to sparse)
° Benchmarks
° Review Gaussian Elimination (GE) for solving Ax=b
° Optimizing GE for caches on sequential machines
  • using matrix-matrix multiplication (BLAS)
° LAPACK library overview and performance
° Data layouts on parallel machines
° Parallel matrix-matrix multiplication review
° Parallel Gaussian Elimination
° ScaLAPACK library overview
° Eigenvalue problems
° Open Problems
Motivation

3 Basic Linear Algebra Problems

- Linear Equations: Solve $Ax = b$ for $x$
- Least Squares: Find $x$ that minimizes $\sum r_i^2$ where $r = Ax - b$
- Eigenvalues: Find $\lambda$ and $x$ where $Ax = \lambda x$
- Lots of variations depending on structure of $A$ (e.g., symmetry)

Why dense $A$, as opposed to sparse $A$?

- Aren’t “most” large matrices sparse?
- Dense algorithms easier to understand
- Some applications yields large dense matrices
  - $Ax = b$: Computational Electromagnetics
  - $Ax = \lambda x$: Quantum Chemistry
- Benchmarking
  - “How fast is your computer?” = “How fast can you solve dense $Ax = b$?”
- Large sparse matrix algorithms often yield smaller (but still large) dense problems
Computational Electromagnetics – Solve $Ax=b$

• Developed during 1980s, driven by defense applications
• Determine the RCS (radar cross section) of airplane
• Reduce signature of plane (stealth technology)
• Other applications are antenna design, medical equipment

• Two fundamental numerical approaches:
  • MOM methods of moments (frequency domain)
    • Large dense matrices
  • Finite differences (time domain)
    • Even larger sparse matrices
Computational Electromagnetics

- Discretize surface into triangular facets using standard modeling tools

- Amplitude of currents on surface are unknowns

- Integral equation is discretized into a set of linear equations

image: NW Univ. Comp. Electromagnetics Laboratory
After discretization the integral equation has the form

\[ A \ x = b \]

where

- \( A \) is the (dense) impedance matrix,
- \( x \) is the unknown vector of amplitudes, and
- \( b \) is the excitation vector.

(see Cwik, Patterson, and Scott, Electromagnetic Scattering on the Intel Touchstone Delta, IEEE Supercomputing '92, pp 538 - 542)
### Results for Parallel Implementation on Intel Delta

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill (compute $n^2$ matrix entries)</td>
<td>9.20</td>
</tr>
<tr>
<td>(embarrassingly parallel but slow)</td>
<td></td>
</tr>
<tr>
<td>Factor (Gaussian Elimination, $O(n^3)$)</td>
<td>8.25</td>
</tr>
<tr>
<td>(good parallelism with right algorithm)</td>
<td></td>
</tr>
<tr>
<td>Solve ($O(n^2)$)</td>
<td>2.17</td>
</tr>
<tr>
<td>(reasonable parallelism with right algorithm)</td>
<td></td>
</tr>
<tr>
<td>Field Calc. ($O(n)$)</td>
<td>0.12</td>
</tr>
<tr>
<td>(embarrassingly parallel and fast)</td>
<td></td>
</tr>
</tbody>
</table>

The problem solved was for a matrix of size 48,672.

# Current Records for Solving Dense Systems

www.netlib.org, click on Performance DataBase Server

<table>
<thead>
<tr>
<th>Year</th>
<th>System Size</th>
<th>Machine</th>
<th># Procs</th>
<th>Gflops (Peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950's</td>
<td>O(100)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1995</td>
<td>128,600</td>
<td>Intel Paragon</td>
<td>6768</td>
<td>281</td>
</tr>
<tr>
<td>1996</td>
<td>215,000</td>
<td>Intel ASCI Red</td>
<td>7264</td>
<td>1068</td>
</tr>
<tr>
<td>1998</td>
<td>148,000</td>
<td>Cray T3E</td>
<td>1488</td>
<td>1127</td>
</tr>
<tr>
<td>1998</td>
<td>235,000</td>
<td>Intel ASCI Red</td>
<td>9152</td>
<td>1338</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(200 MHz Ppro)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>374,000</td>
<td>SGI ASCI Blue</td>
<td>5040</td>
<td>1608</td>
</tr>
<tr>
<td>2000</td>
<td>362,000</td>
<td>Intel ASCI Red</td>
<td>9632</td>
<td>2380</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(333 MHz Xeon)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>518,000</td>
<td>IBM ASCI White</td>
<td>8000</td>
<td>7226</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(375 MHz Power 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

# Current Records for Solving Small Dense Systems

[www.netlib.org](http://www.netlib.org), click on Performance DataBase Server

<table>
<thead>
<tr>
<th>Machine</th>
<th>Megaflops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n=100</td>
</tr>
<tr>
<td>Fujitsu VPP 5000</td>
<td>1156</td>
</tr>
<tr>
<td>(1 proc 300 MHz)</td>
<td></td>
</tr>
<tr>
<td>Cray T90</td>
<td></td>
</tr>
<tr>
<td>(32 proc, 450 MHz)</td>
<td>29360</td>
</tr>
<tr>
<td>(4 proc, 450 MHz)</td>
<td>1129</td>
</tr>
<tr>
<td>IBM Power 4</td>
<td></td>
</tr>
<tr>
<td>(1 proc, 1300 MHz)</td>
<td>1074</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Dell Itanium</td>
<td></td>
</tr>
<tr>
<td>(4 proc, 800 MHz)</td>
<td>7358</td>
</tr>
<tr>
<td>(2 proc, 800 MHz)</td>
<td>4504</td>
</tr>
<tr>
<td>(1 proc, 800 MHz)</td>
<td>600</td>
</tr>
</tbody>
</table>

Computational Chemistry – $Ax = \lambda x$

- Seek energy levels of a molecule, crystal, etc.
  - Solve Schrödinger’s Equation for energy levels = eigenvalues
  - Discretize to get $Ax = \lambda Bx$, solve for eigenvalues $\lambda$ and eigenvectors $x$
  - $A$ and $B$ large, symmetric or Hermitian matrices ($B$ positive definite)
  - May want some or all eigenvalues and/or eigenvectors

- MP-Quest (Sandia NL)
  - Si and sapphire crystals of up to 3072 atoms
  - Local Density Approximation to Schrödinger Equation
  - $A$ and $B$ up to $n=40000$, complex Hermitian
  - Need all eigenvalues and eigenvectors
  - Need to iterate up to 20 times (for self-consistency)

- Implemented on Intel ASCI Red
  - 9200 Pentium Pro 200 processors (4600 Duals, a CLUMP)
  - Overall application ran at 605 Gflops (out of 1800 Gflops peak),
  - Eigensolver ran at 684 Gflops
  - Runner-up for Gordon Bell Prize at Supercomputing 98

- Same problem arises in astrophysics...
Review of Gaussian Elimination (GE) for solving $Ax=b$

- Add multiples of each row to later rows to make $A$ upper triangular

- Solve resulting triangular system $Ux = c$ by substitution

```
... for each column $i$
... zero it out below the diagonal by adding multiples of row $i$ to later rows
for $i = 1$ to $n-1$
    ... for each row $j$ below row $i$
    for $j = i+1$ to $n$
        ... add a multiple of row $i$ to row $j$
        for $k = i$ to $n$
            $A(j,k) = A(j,k) - (A(j,i)/A(i,i)) \cdot A(i,k)$
```

Structure of Matrix during simple version of Gaussian Elimination

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

After $i=1$

```
0 0 0 0
0 0 0 0
0 0 0 0
```

After $i=2$

```
0 0 0 0
0 0 0 0
0 0 0 0
```

After $i=3$

```
0 0 0 0
0 0 0 0
0 0 0 0
```

After $i=n-1$
Refine GE Algorithm (1)

° Initial Version

... for each column i
... zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
  ... for each row j below row i
  for j = i+1 to n
    ... add a multiple of row i to row j
    for k = i to n
      A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)

° Remove computation of constant A(j,i)/A(i,i) from inner loop

for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i to n
      A(j,k) = A(j,k) - m * A(i,k)
Refine GE Algorithm (2)

° Last version

```plaintext
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i to n
      A(j,k) = A(j,k) - m * A(i,k)
```

° Don’t compute what we already know: zeros below diagonal in column i

```plaintext
for i = 1 to n-1
  for j = i+1 to n
    m = A(j,i)/A(i,i)
    for k = i+1 to n
      A(j,k) = A(j,k) - m * A(i,k)
```
Refine GE Algorithm (3)

° Last version

for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - m * A(i,k)

° Store multipliers m below diagonal in zeroed entries for later use

for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
Refine GE Algorithm (4)

° Last version

for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i)/A(i,i)
    for k = i+1 to n
      A(j,k) = A(j,k) - A(j,i) * A(i,k)

° Split Loop

for i = 1 to n-1
  for j = i+1 to n
    A(j,i) = A(j,i)/A(i,i)
    for j = i+1 to n
      for k = i+1 to n
        A(j,k) = A(j,k) - A(j,i) * A(i,k)
Refine GE Algorithm (5)

° Last version

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{for } j = i+1 \text{ to } n \\
A(j,i) &= A(j,i)/A(i,i) \\
\text{for } j = i+1 \text{ to } n \\
\text{for } k = i+1 \text{ to } n \\
A(j,k) &= A(j,k) - A(j,i) \cdot A(i,k)
\end{align*}
\]

° Express using matrix operations (BLAS)

Work at step i of Gaussian Elimination

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
A(i+1:n,i) &= A(i+1:n,i) \cdot (1 / A(i,i)) \\
A(i+1:n,i+1:n) &= A(i+1:n,i+1:n) - A(i+1:n,i) \cdot A(i,i+1:n)
\end{align*}
\]
What GE really computes

for i = 1 to n-1
  A(i+1:n,i) = A(i+1:n,i) / A(i,i)
  A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n)

° Call the strictly lower triangular matrix of multipliers M, and let L = I+M

° Call the upper triangle of the final matrix U

° Lemma (LU Factorization): If the above algorithm terminates (does not divide by zero) then A = L*U

° Solving A*x=b using GE
  • Factorize A = L*U using GE (cost = 2/3 n^3 flops)
  • Solve L*y = b for y, using substitution (cost = n^2 flops)
  • Solve U*x = y for x, using substitution (cost = n^2 flops)

° Thus A*x = (L*U)*x = L*(U*x) = L*y = b as desired
Problems with basic GE algorithm

- What if some $A(i,i)$ is zero? Or very small?
  - Result may not exist, or be “unstable”, so need to **pivot**

- Current computation all BLAS 1 or BLAS 2, but we know that **BLAS 3** (matrix multiply) is fastest (earlier lectures...)

```
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)
```
Pivoting in Gaussian Elimination

° A = [ 0 1 ] fails completely, even though A is “easy”
   [ 1 0 ]

° Illustrate problems in 3-decimal digit arithmetic:

A = [ 1e-4 1 ] and b = [ 1 ], correct answer to 3 places is x = [ 1 ]
   [ 1 1 ] [ 2 ]

° Result of LU decomposition is

L = [ 1 0 ] = [ 1 0 ] … No roundoff error yet
   [ fl(1/1e-4) 1 ] [ 1e4 1 ]

U = [ 1e-4 1 ] = [ 1e-4 1 ] … Error in 4th decimal place
   [ 0 fl(1-1e4*1) ] [ 0 -1e4 ]

Check if A = L*U = [ 1e-4 1 ] … (2,2) entry entirely wrong
   [ 1 0 ]

° Algorithm “forgets” (2,2) entry, gets same L and U for all |A(2,2)|<5
  ° Numerical instability
  ° Computed solution x totally inaccurate
  ° Cure: Pivot (swap rows of A) so entries of L and U bounded
Gaussian Elimination with Partial Pivoting (GEPP)

° Partial Pivoting: swap rows so that each multiplier
   \[ |L(i,j)| = \frac{|A(j,i)|}{A(i,i)} \leq 1 \]

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) = \frac{A(i+1:n,i)}{A(i,i)} \quad \text{... each quotient lies in \([-1,1]\)}
\]
\[
A(i+1:n,i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i) \times A(i, i+1:n)
\]

° Lemma: This algorithm computes \( A = P*L*U \), where \( P \) is a
  permutation matrix

° Since each entry of \( |L(i,j)| \leq 1 \), this algorithm is considered
  numerically stable

° For details see LAPACK code at www.netlib.org/lapack/single/sgetf2.f
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
  - 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
Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

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 \( b \)

(For a correctness proof, see on-lines notes.)
Efficiency of Blocked GEPP

![Graph showing efficiency of LAPACK LU for n=1000 across different systems and processor counts.](image-url)