
CS 267 Applications of Parallel Processors

Lecture 9: Computational Electromagnetics - Large Dense Linear Systems

2/19/97

Horst D. Simon

<http://www.cs.berkeley.edu/cs267>

Outline - Lecture 9

- **Computational Electromagnetics**
- **Sources of large dense linear systems**
- **Review of solution of linear systems with Gaussian elimination**
- **BLAS and memory hierarchy for linear algebra kernels**

Outline - Lecture 10

- **Layout of matrices on distributed memory machines**
- **Distributed Gaussian elimination**
- **Speeding up with advanced algorithms**
- **LINPACK and LAPACK**
- **LINPACK benchmark**
- **Tflops result**

Outline - Lecture 11

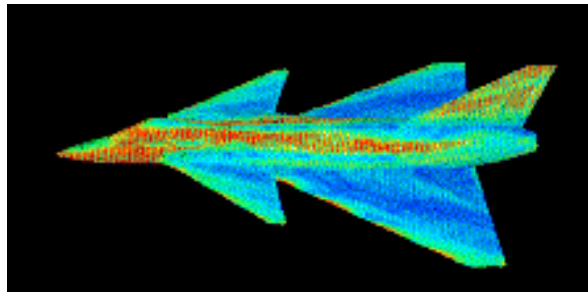
- Designing portable libraries for parallel machines
- BLACS
- ScaLAPACK for dense linear systems
- other linear algebra algorithms in ScaLAPACK

Computational Electromagnetics

- 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), and finite differences (time domain)

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 <http://nueml.ece.nwu.edu/>

Computational Electromagnetics (MOM)

After discretization the integral equation has the form

$$\mathbf{Z} \mathbf{J} = \mathbf{V}$$

where

\mathbf{Z} is the impedance matrix, \mathbf{J} is the unknown vector of amplitudes, and \mathbf{V} is the excitation vector.

\mathbf{Z} is given as a four dimensional integral.

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

Computational Electromagnetics (MOM)

The main steps in the solution process are

- A) computing the matrix elements
- B) factoring the dense matrix
- C) solving for one or more excitations
- D) computing the fields scattered from the object

Analysis of MOM for Parallel Implementation

Task	Work	Parallelism	Parallel Speed
Fill	$O(n^{**2})$	embarrassing	low
→ Factor	$O(n^{**3})$	moderately diff.	very high
Solve	$O(n^{**2})$	moderately diff.	high
Field Calc.	$O(n)$	embarrassing	high

For most scientific applications the biggest gain in performance can be obtained by focusing on one tasks.

Results for Parallel Implementation on Delta

Task	Time (hours)	Performance (Gflop/s)
Fill	9.20	~ 1.0
Factor	8.25	10.35
Solve	2.17	-
Field Calc.	0.12	3.0

The problem solved was for a matrix of size 48,672. (The world record in 1991.)

Current Records for Solving Dense Systems

Year	System Size	Machine
1950's	O(100)	
1991	55,296	CM-2
1992	75,264	Intel
1993	75,264	Intel
1994	76,800	CM-5
1995	128,600	Paragon XP
1996	215,000	ASCI Red

source: Alan Edelman <http://www-math.mit.edu/~edelman/records.html>

Sources for large dense linear systems

- not many outside CEM
- even within CEM community alternatives such FD-TD are heavily debated

In many instances choices for algorithms or methods in existing scientific codes or applications are not the result of careful planning and design. At best they are reflecting the start-of-the-art at the time, at worst they are purely coincidental.

Review of Gaussian Elimination

Gaussian elimination to solve $Ax=b$

- start with a dense matrix
 - add multiples of each row to subsequent rows in order to create zeros below the diagonal
 - ending up with an upper triangular matrix U .
- Solve a linear system with U by substitution, starting with the last variable.**

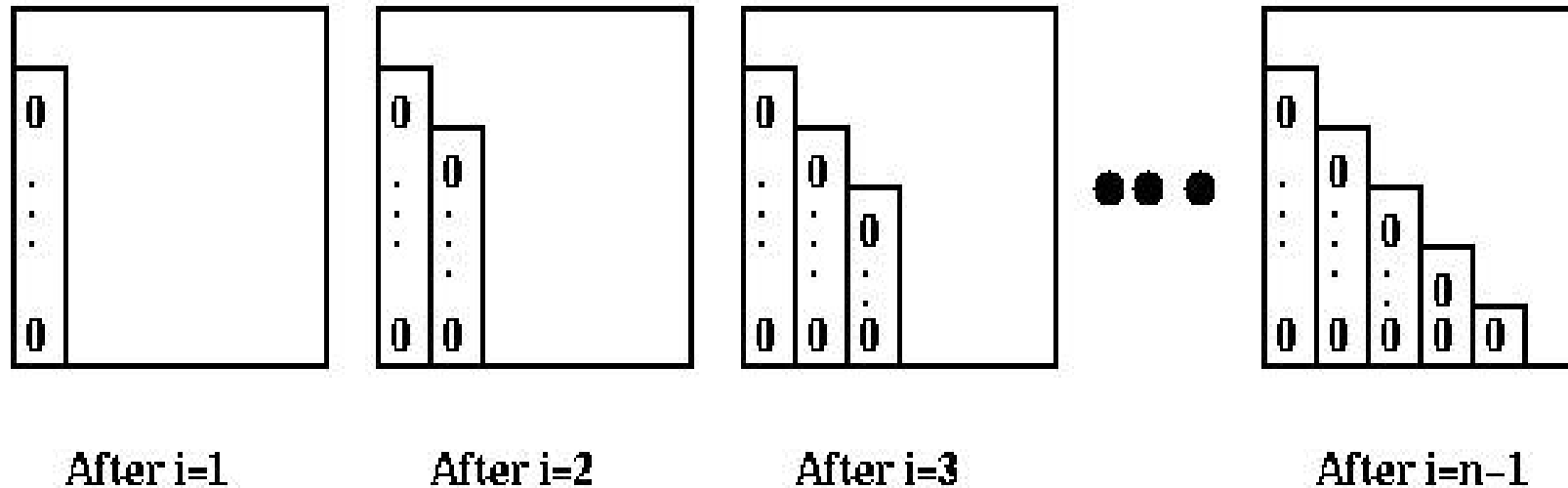
see Demmel <http://HTTP.CS.Berkeley.EDU/~demmel/cs267/lecture12/lecture12.html>

Review of Gaussian Elimination (cont.)

```
... 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  
    ... 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  
             $A(j,k) = A(j,k) -$   
                 $(A(j,i)/A(i,i)) * A(i,k)$ 
```

Review of Gaussian Elimination (cont.)

Structure of Matrix during simple version of Gaussian Elimination

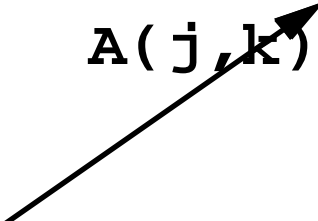


Review of Gaussian Elimination (cont.)

```
... 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  
  ... 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  
       $A(j,k) = A(j,k) -$   
         $(A(j,i)/A(i,i)) * A(i,k)$   
      = m
```

Review of Gaussian Elimination (cont.)

```
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)
```



avoid computation of known matrix entry

Review of Gaussian Elimination (cont.)

It will be convenient to store the multipliers m in the implicitly created zeros below the diagonal, so we can use them later to transform the right hand side b :

```
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)
```

Review of Gaussian Elimination (cont.)

Now we use Matlab (data parallel) notation to express the algorithm even more compactly:

```
for i = 1 to n-1
```

```
    A(i+1:n, i) = A(i+1:n, i) / A(i, i)
```

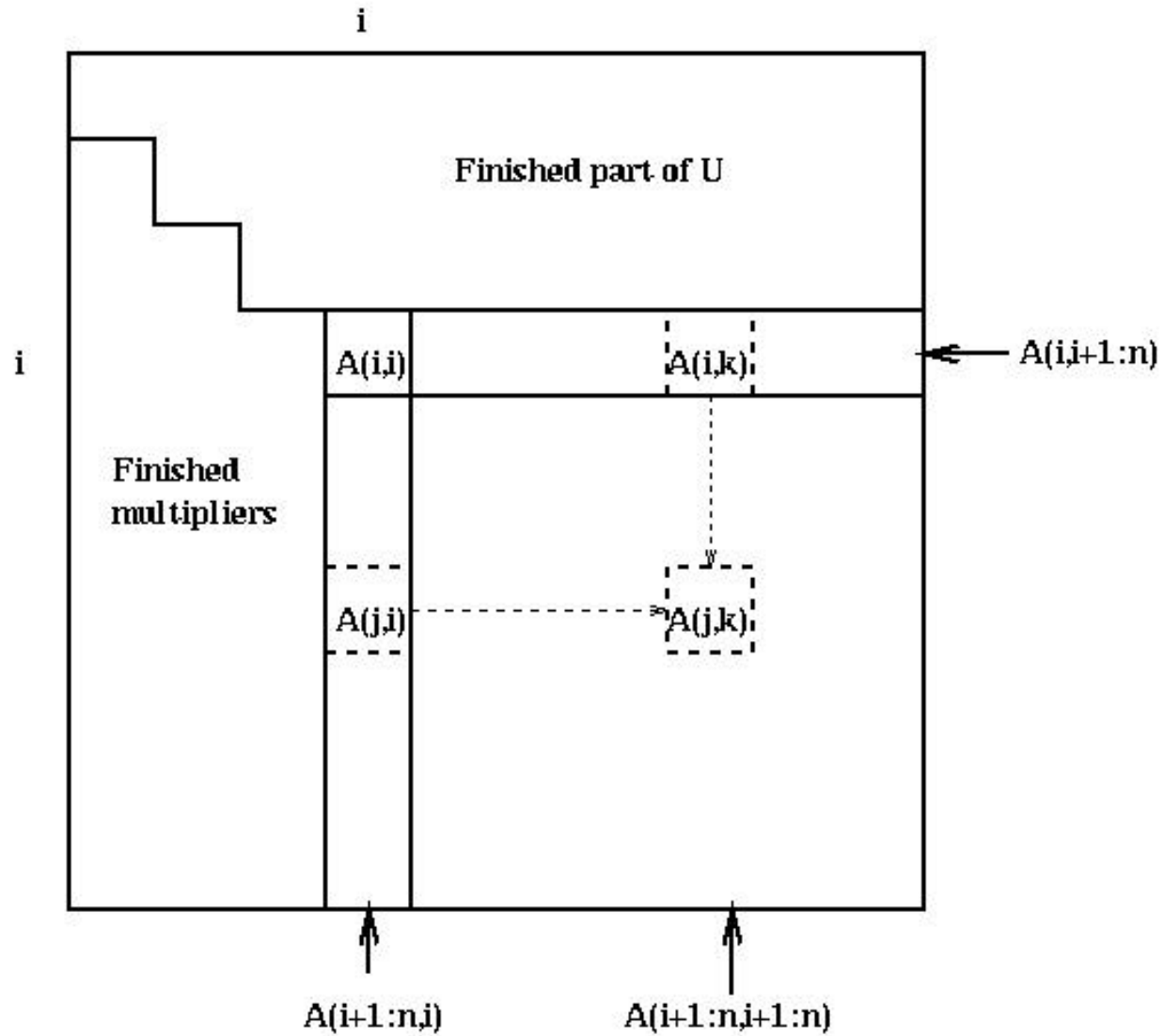
```
    A(i+1:n, i+1:n) = A(i+1:n, i+1:n) -  
                    A(i+1:n, i)*A(i, i+1:n)
```

The inner loop consists of one vector operation, and one matrix-vector operation.

Note that the loop looks elegant, but no longer intuitive.

Review of Gaussian Elimination (cont.)

Work at step 1 of Gaussian Elimination



Review of Gaussian Elimination (cont.)

Lemma. (LU Factorization). If the above algorithm terminates (i.e. it did not try to divide by zero) then $A = L*U$.

Now we can state our complete algorithm for solving $A*x=b$:

- 1) Factorize $A = L*U$.
- 2) Solve $L*y = b$ for y
by forward substitution.
- 3) Solve $U*x = y$ for x
by backward substitution.

Then x is the solution we seek because $A*x = L*(U*x) = L*y = b$.

Review of Gaussian Elimination (cont.)

Here are some obvious problems with this algorithm, which we need to address:

- If $A(i,i)$ is zero, the algorithm cannot proceed.
If $A(i,i)$ is tiny, we will also have numerical problems.
- The majority of the work is done by a rank-one update, which does not exploit a memory hierarchy as well as an operation like matrix-matrix multiplication

Pivoting for Small $A(i,i)$

Why pivoting is needed?

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Even if $A(i,i)$ is tiny, but not zero difficulties can arise (see example in Jim Demmel's lecture notes).

This problem is resolved by *partial pivoting*.

Partial Pivoting

Reordering the rows of A so that $A(i,i)$ is large at each step of the algorithm.

At step i of the algorithm, row i is swapped with row $k > i$ if $|A(k,i)|$ is the largest entry among $|A(i:n,i)|$.

```
for i = 1 to n-1
  find and record k where
     $|A(k,i)| = \max_{i \leq j \leq n} |A(j,i)|$ 
  if  $|A(k,i)| = 0$ , exit with a warning
    that  $A$  is singular, or nearly so
  if  $i \neq k$ , swap rows  $i$  and  $k$  of  $A$ 
   $A(i+1:n, i) = A(i+1:n, i) / A(i,i)$ 
  ... each quotient lies in  $[-1,1]$ 
   $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) -$ 
     $A(i+1:n, i) * A(i, i+1:n)$ 
```

Partial Pivoting (cont.)

- for 2-by-2 example, we get a very accurate answer
- several choices as to when to swap rows i and k
- could use indirect addressing and not swap them at all, but this would be slow
- keep permutation, then solving $A^*x=b$ only requires the additional step of permuting b

Fast linear algebra kernels: BLAS

- Simple linear algebra kernels such as matrix-matrix multiply (exercise) can be performed fast on memory hierarchies.
- More complicated algorithms can be built from some very basic building blocks and kernels.
- The interfaces of these kernels have been standardized as the Basic Linear Algebra Subroutines or BLAS.
- Early agreement on standard interface (around 1980) led to portable libraries for vector and shared memory parallel machines.
- BLAS are classified into three categories, level 1,2,3

see Demmel <http://HTTP.CS.Berkeley.EDU/~demmel/cs267/lecture02.html>

Level 1 BLAS

Operate mostly on vectors (1D arrays), or pairs of vectors; perform $O(n)$ operations; return either a vector or a scalar.

Examples

saxpy $y(i) = a * x(i) + y(i)$, for $i=1$ to n .

Saxpy is an acronym for the operation. **S** stands for single precision, **daxpy** is for double precision, **caxpy** for complex, and **zaxpy** for double complex,

sscal $y = a * x$,

srot replaces vectors x and y by $c*x+s*y$ and $-s*x+c*y$, where c and s are typically a cosine and sine.

sdot computes $s = \sum_{i=1}^n x(i)*y(i)$

Level 2 BLAS

operate mostly on a matrix (2D array) and a vector;
return a matrix or a vector; $O(n^2)$ operations.

Examples.

sgemv Matrix-vector multiplication computes $y = y + A*x$
where A is m -by- n , x is n -by-1 and y is m -by-1.

sger rank-one update computes $A = A + y*x'$, where A
is m -by- n , y is m -by-1, x is n -by-1, x' is the transpose
of x . This is a short way of saying $A(i,j) = A(i,j) + y(i)*x(j)$
for all i,j .

strsv triangular solve solves $y=T*x$ for x , where T is a
triangular matrix.

Level 3 BLAS

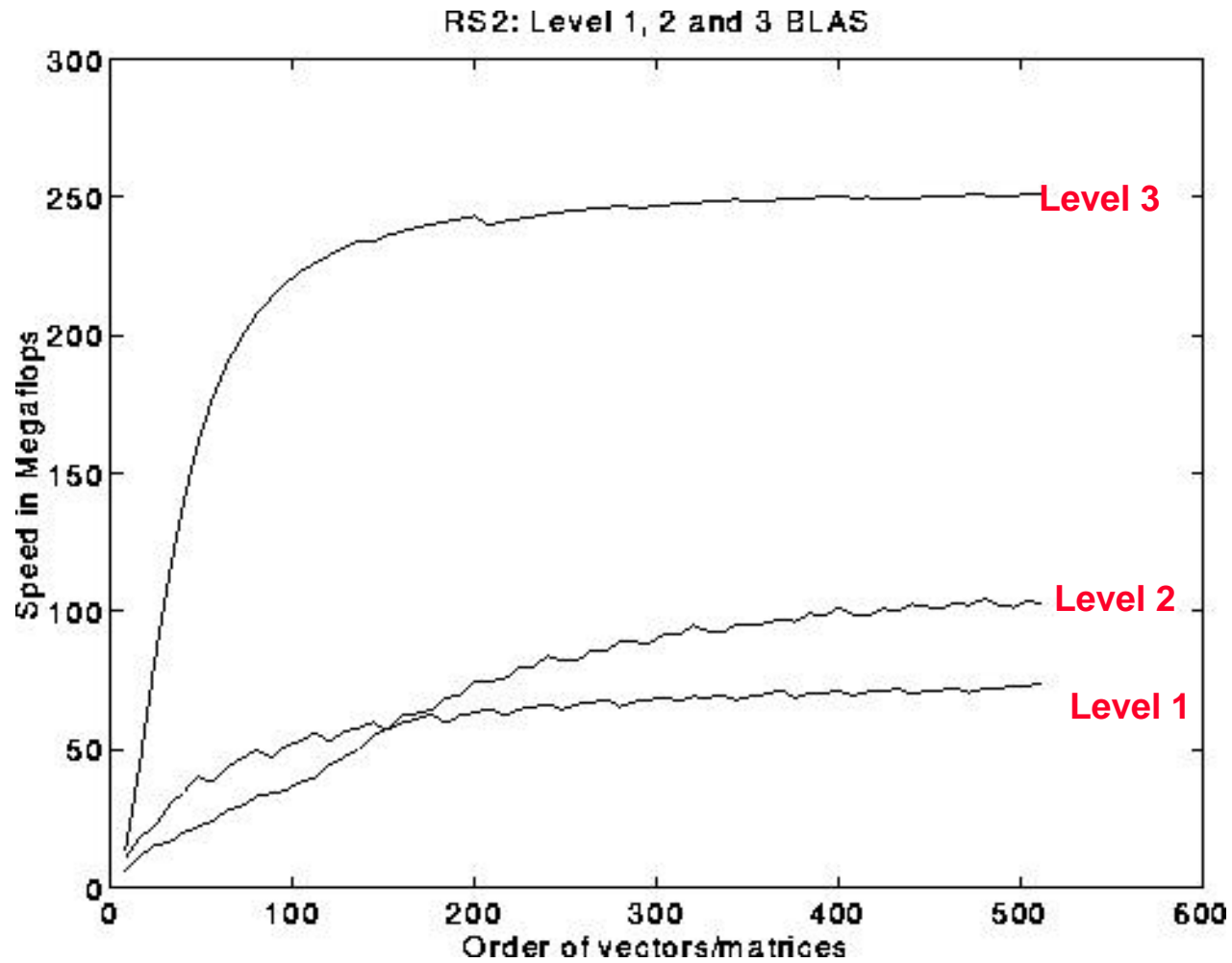
operate on pairs or triples of matrices, returning a matrix; complexity is $O(n^3)$.

Examples

sgemm Matrix-matrix multiplication computes $C = C + A*B$, where C is m -by- n , A is m -by- k , and B is k -by- n

sgtrsm multiple triangular solve solves $Y = T*X$ for X , where T is a triangular matrix, and X is a rectangular matrix.

Performance of BLAS



Performance of BLAS (cont.)

- **BLAS are specially optimized by the vendor (IBM) to take advantage of all features of the RS 6000/590.**
- **Potentially a big speed advantage if an algorithm can be expressed in terms of the BLAS3 instead of BLAS2 or BLAS1.**
- **The top speed of the BLAS3, about 250 Mflops, is very close to the peak machine speed of 266 Mflops.**
- **We will reorganize algorithms, like Gaussian elimination, so that they use BLAS3 rather than BLAS1 or BLAS2.**

Explanation of Performance of BLAS

m = number of memory references to slow memory (read + write)

f = number of floating point operations

q = f/m = average number of flops per slow memory reference

	m	justification for m	f	q
saxpy	$3*n$	read $x(i)$, $y(i)$; write $y(i)$	$2*n$	$2/3$
sgemv	$n^2+O(n)$	read each $A(i,j)$ once	$2*n^2$	2
sgemm	$4*n^2$	read $A(i,j),B(i,j),C(i,j)$ write $C(i,j)$ once	$2*n^3$	$n/2$