Lecture 8 (part 2): Linear Algebra Algorithms

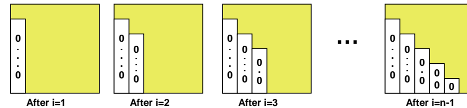
Jack Dongarra, U of Tennessee

Slides are adapted from Jim Demmel, UCB's Lecture on Linear Algebra Algorithms

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
        tmp = A(j,i);
        for k = i to n
            A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```



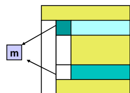
Refine GE Algorithm (1)

- 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
        tmp = A(j,i);
        for k = i to n
            A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

- Remove computation of constant tmp/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)
```



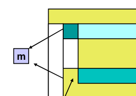
Refine GE Algorithm (2)

- Last version

```
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

```
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)
```



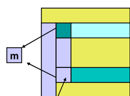
Refine GE Algorithm (3)

- 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)
```



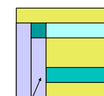
Refine GE Algorithm (4)

- 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)
```

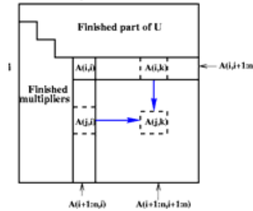


Refine GE Algorithm (5)

- Last version

```
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)
```

- Express using matrix operations (BLAS)




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


What GE really computes

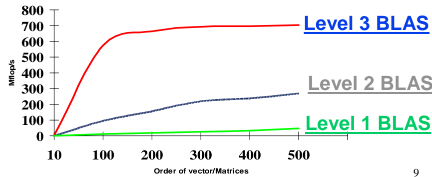
```
for i = 1 to n-1
   A(i+1:n,i) = A(i+1:n,i) / A(i,i)
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- 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 because can't divide by A(1,1)=0
  [1 0]
- But solving Ax=b should be easy!
- When diagonal A(i,i) is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - Numerical instability
  - Roundoff error is cause
- Cure: Pivot (swap rows of A) so A(i,i) large

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that A(i,i) is largest in column

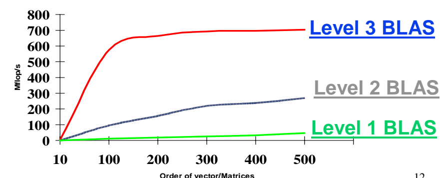
```
for i = 1 to n-1
   find and record k where |A(k,i)| = max{i <= j <= n} |A(j,i)|
      … i.e. largest entry in rest of column i
   if |A(k,i)| = 0
      exit with a warning that A is singular, or nearly so
   elseif k != i
      swap rows i and k of A
   end if
   A(i+1:n,i) = A(i+1:n,i) / A(i,i)       … each quotient lies in [-1,1]
   A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- *Lemma*: This algorithm computes A = P*L*U, where P is a permutation matrix.
- This algorithm is numerically stable in practice
- For details see LAPACK code at http://www.netlib.org/lapack/single/sgetf2.f

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)
```

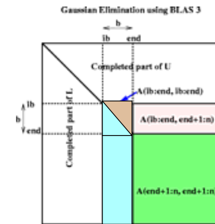


Converting BLAS2 to BLAS3 in GEPP

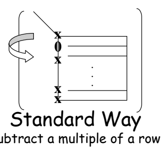
- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- BIG IDEA: Delayed Updates
  - Save updates to "trailing matrix" from several consecutive BLAS2 updates
  - Apply many updates simultaneously in one BLAS3 operation
- Same idea works for much of dense linear algebra
  - Open questions remain
- First Approach: 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

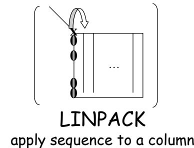
Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

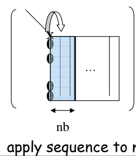
```
for  ib = 1 to n-1 step b      … Process matrix b columns at a time
   end = ib + b-1              … Point to end of block of b columns
   apply BLAS2 version of GEPP to get A(ib:n , ib:end) = P' * L' * U'
   … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
   A(ib:end , 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
```



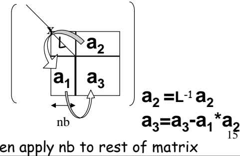
Gaussian Elimination







LAPACK

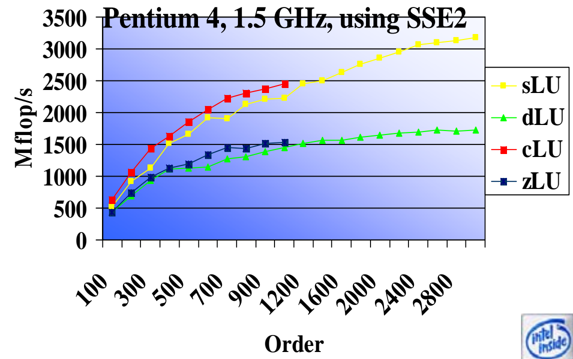


$a_2 = L^{-1} a_2$

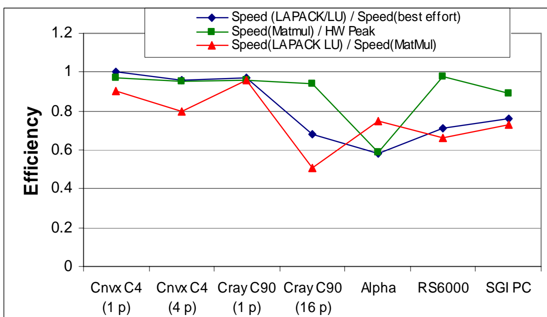
$a_3 = a_3 - a_1 * a_2$

LU Factorization

Pentium 4, 1.5 GHz, using SSE2



Efficiency of Blocked GEPP



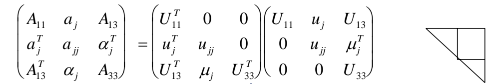
History of Block Partitioned Algorithms

- Early algorithms involved use of small main memory using tapes as secondary storage.
- Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and "out of core" memory.

Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or Q^TC
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems

Derivation of Blocked Algorithms
Cholesky Factorization $A = U^TU$



Equating coefficient of the jth column, we obtain

$$a_j = U_{11}^T u_j$$
$$a_{jj} = u_j^T u_j + u_{jj}^2$$

Hence, if U_{11} has already been computed, we can compute u_j and u_{jj} from the equations:

$$U_{11}^T u_j = a_j$$
$$u_{jj}^2 = a_{jj} - u_j^T u_j$$

LINPACK Implementation

- Here is the body of the LINPACK routine SPOFA which implements the method:

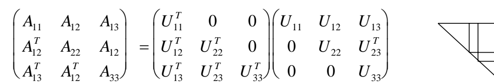
```
      DO 30 J = 1, N
         INFO = J
         S = 0.0E0
         JM1 = J - 1
         IF( JM1.LT.1 ) GO TO 20
         DO 10 K = 1, JM1
            T = A( K, J ) - SDOT( K-1, A( 1, K ), 1,A( 1, J ), 1 )
            T = T / A( K, K )
            A( K, J ) = T
            S = S + T*T
   10    CONTINUE
   20    CONTINUE
         S = A( J, J ) - S
C     ...EXIT
         IF( S.LE.0.0E0 ) GO TO 40
         A( J, J ) = SQRT( S )
   30 CONTINUE
```


LAPACK Implementation

```
   DO 10 J = 1, N
      CALL STRSV( 'Upper', 'Transpose', 'Non-Unit', J-1, A, LDA, A( 1, J ), 1 )
      S = A( J, J ) - SDOT( J-1, A( 1, J ), 1, A( 1, J ), 1 )
      IF( S.LE.ZERO ) GO TO 20
      A( J, J ) = SQRT( S )
10 CONTINUE
```

- This change by itself is sufficient to significantly improve the performance on a number of machines.
- From 238 to 312 Mflop/s for a matrix of order 500 on a Pentium 4-1.7 GHz.
- However on peak is 1,700 Mflop/s.
- Suggest further work needed.

Derivation of Blocked Algorithms



Equating coefficient of second block of columns, we obtain

$$A_{12} = U_{11}^T U_{12}$$
$$A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}$$

Hence, if U_{11} has already been computed, we can compute U_{12} as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

$$U_{11}^T U_{12} = A_{12}$$
$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

LAPACK Blocked Algorithms

```
   DO 10 J = 1, N, NB
      CALL STRSM( 'Left', 'Upper', 'Transpose','Non-Unit', J-1, JB, ONE, A, LDA,
$              A( 1, J ), LDA )
      CALL SSYRK( 'Upper', 'Transpose', JB, J-1,-ONE, A( 1, J ), LDA, ONE,
$              A( J, J ), LDA )
      CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
      IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

- On Pentium 4, L3 BLAS squeezes a lot more out of 1 proc

Intel Pentium 4 1.7 GHz N = 500	Rate of Execution
Linpack variant (L1B)	238 Mflop/s
Level 2 BLAS Variant	312 Mflop/s
Level 3 BLAS Variant	1262 Mflop/s

Overview of LAPACK and ScaLAPACK

- Standard library for dense/banded linear algebra
  - Linear systems: A*x=b
  - Least squares problems: $\min_x || A*x-b ||_2$
  - 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, Matlab …
- Many algorithmic innovations remain
  - Projects available

LAPACK

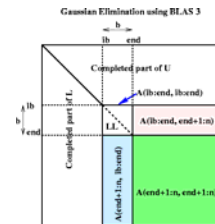
- Linear Algebra library in Fortran 77
  - Solution of systems of equations
  - Solution of eigenvalue problems
- Combine algorithms from LINPACK and EISPACK into a single package
- Efficient on a wide range of computers
  - RISC, Vector, SMPs
- User interface similar to LINPACK
  - Single, Double, Complex, Double Complex
- Built on the Level 1, 2, and 3 BLAS

LAPACK

- Most of the parallelism in the BLAS.
- Advantages of using the BLAS for parallelism:
  - Clarity
  - Modularity
  - Performance
  - Portability

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
BLAS 3 { 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
```



Parallelizing Gaussian Elimination

- Parallelization steps
  - 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
  - Need to use parallel matrix multiplication
- Assignment
  - Which processors are responsible for which submatrices?

Challenges in Developing Distributed Memory Libraries

- How to integrate software?
  - Until recently no standards
  - Many parallel languages
  - Various parallel programming models
  - Assumptions about the parallel environment
    - granularity
    - topology
    - overlapping of communication/computation
    - development tools
- Where is the data
  - Who owns it?
  - Opt data distribution
- Who determines data layout
  - Determined by user?
  - Determined by library developer?
  - Allow dynamic data dist.
  - Load balancing

Different Data Layouts for Parallel GE

Bad load balance: P0 idle after first n/4 steps

1) 1D Column Blocked Layout

Load balanced, but can't easily use BLAS2 or BLAS3

2) 1D Column Cyclic Layout

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

3) 1D Column Block Cyclic Layout

Complicated addressing

4) Block Skewed Layout

Bad load balance: P0 idle after first n/2 steps

5) 2D Row and Column Blocked Layout

The winner!

6) 2D Row and Column Block Cyclic Layout

Review of Parallel MatMul

- Want Large Problem Size Per Processor

PDGEMM = PBLAS matrix multiply

Observations:

- For fixed N, as P increasesn 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:

- Efficiency always at least 48%
- For fixed N, as P increases, efficiency drops
- For fixed P, as N increases, efficiency increases

Performance of PBLAS

Speed in Mflops of PDGEMM

Machine	Procs	Block Size	N = 2000	N = 4000	N = 10000
Cray T3E	4=2x2	32	1055	1070	0
	16=4x4		3630	4005	4292
	64=8x8		13456	14287	16755
IBM SP2	4	50	755	0	0
	16		2514	2850	0
	64		6205	8709	10774
Intel XP/S MP Paragon	4	32	330	0	0
	16		1233	1281	0
	64		4496	4864	5257
Berkeley NOW	4	32	463	470	0
	32=4x8		2490	2822	3450
	64		4130	5457	6647

Efficiency = MFlops(PDGEMM)/(Procs*MFlops(DGEMM))

Machine	Peak/proc	DGEMM Mflops	Procs	N = 2000	N = 4000	N = 10000
Cray T3E	600	360	4	.73	.74	
			16	.63	.70	.75
			64	.58	.62	.73
IBM SP2	266	200	4	.94		
			16	.79	.89	
			64	.48	.68	.84
Intel XP/S MP Paragon	100	90	4	.92		
			16	.86	.89	
			64	.78	.84	.91
Berkeley NOW	334	129	4	.90	.91	
			32	.60	.68	.84
			64	.50	.66	.81

Row and Column Block Cyclic Layout

bcol

brow

- processors and matrix blocks are distributed in a 2d array
  - prow-by-pcol array of processors
  - brow-by-bcol matrix blocks
- pcol-fold parallelism in any column, and calls to the BLAS2 and BLAS3 on matrices of size brow-by-bcol
- serial bottleneck is eased
- prow ≠ pcol and brow ≠ bcol possible, even desireable

Distributed GE with a 2D Block Cyclic Layout

- block size b in the algorithm and the block sizes brow and bcol in the layout satisfy b=bcol.
- shaded regions indicate processors busy with computation or communication.
- unnecessary to have a barrier between each step of the algorithm, e.g.. step 9, 10, and 11 can be pipelined

ScaLAPACK

- Library of software dealing with dense & banded routines
- Distributed Memory - Message Passing
- MIMD Computers and Networks of Workstations
- Clusters of SMPs

Programming Style

- SPMD Fortran 77 with object based design
- Built on various modules
  - PBLAS Interprocessor communication
  - BLACS
    - PVM, MPI, IBM SP, CRI T3, Intel, TMC
    - Provides right level of notation.
  - BLAS
- LAPACK software expertise/quality
  - Software approach
  - Numerical methods

Overall Structure of Software

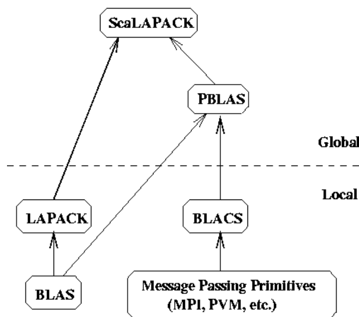
- Object based - Array descriptor
  - Contains information required to establish mapping between a global array entry and its corresponding process and memory location.
  - Provides a flexible framework to easily specify additional data distributions or matrix types.
  - Currently dense, banded, & out-of-core
- Using the concept of context

PBLAS

- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix

```
CALL DGEXXX  ( M, N, A( IA, JA ), LDA,... )
        ↓
CALL PDGEXXX( M, N, A, IA, JA, DESCA,... )
```


ScaLAPACK Overview



ScaLAPACK Performance Models (1)

ScaLAPACK Operation Counts

$$T(N,P) = \frac{C_f N^3}{P} t_f + \frac{C_v N^2}{\sqrt{P}} t_v + \frac{C_m N}{NB} t_m, \qquad T_{seq}(N,P) = C_f N^3 t_f.$$

$$E(N,P) = \left(1 + \frac{1}{NB}\frac{C_m t_m}{C_f t_f}\frac{P}{N^2} + \frac{C_v t_v}{C_f t_f}\frac{\sqrt{P}}{N}\right)^{-1}.$$

$t_f = 1$
$t_m = \alpha$
$t_v = \beta$
NB = brow=bcol
√P = prow = pcol

Driver	Options	C_f	C_v	C_m
PxGESV	1 right hand side	2/3	$3+1/4\log_2 P$	$NB(6+\log_2 P)$
PxPOSV	1 right hand side	1/3	$2+1/2\log_2 P$	$4+\log_2 P$
PxGELS	1 right hand side	4/3	$3+\log_2 P$	$2(NB\log_2 P+1)$
PxSYEVX	eigenvalues only	4/3	$5/2\log_2 P$	$17/2\,NB+2$
PxSYEVX	eigenvalues and eigenvectors	10/3	$5\log_2 P$	$17/2\,NB+2$
PxSYEV	eigenvalues only	4/3	$5/2\log_2 P$	$17/2\,NB+2$
PxSYEV	eigenvalues and eigenvectors	22/3	$5\log_2 P$	$17/2\,NB+2$
PxGESVD	singular values only	26/3	$10\log_2 P$	$17NB$
PxGESVD	singular values and left and right singular vectors	38/3	$14\log_2 P$	$17NB$
PxLAHQR	eigenvalues only	5	$9/2(\sqrt{P})*\log_2 P + 3N/NB$	$9(2+\log_2 P)N$
PxLAHQR	full Schur form	18	$9/2(\sqrt{P})*\log_2 P + 3N/NB$	$9(2+\log_2 P)N$

ScaLAPACK Performance Models (2)

Compare Predictions and Measurements

IBM SP2[a]	P	Values of N									
		2000		5000		7500		10000		15000	
		Est	Obt	Est	Obt	Est	Obt	Est	Obt	Est	Obt
PDGESV	4	357	421	632	603						
(LU)	16	497	722	1581	1543	2116	1903	2424	2149		
	64	502	924	2432	3017	4235	4295	5793	5596	7992	7057
PDPOSV	4	530	462	669	615						
(Cholesky)	16	1315	1081	2083	1811	2366	2118	2535	2312		
	64	2577	1807	5327	4431	6709	5727	7661	6826	8887	8084

[a] One process spawned per node and one computational IBM POWER2 590 processor per node.

PDGESV = ScaLAPACK Parallel LU

Since it can run no faster than its inner loop (PDGEMM), we measure:
Efficiency =
Speed(PDGESV)/Speed(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 $(2*n^3)/(2/3*n^3) = 3$ times faster, but communication makes it a little slower.

Performance of ScaLAPACK LU

Efficiency = MFlops(PDGESV)/MFlops(PDGEMM)					
Machine	Procs	Block Size	N = 2000	N = 4000	N = 10000
Cray T3E	4	32	.67	.82	
	16		.44	.65	.84
	64		.18	.47	.75
IBM SP2	4	50	.56		
	16		.29	.52	
	64		.15	.32	.66
Intel XP/S MP Paragon	4	32	.64		
	16		.37	.66	
	64		.16	.42	.75
Berkeley NOW	4	32	.76		
	32		.38	.62	.71
	64		.28	.54	.69

Time(PDGESV)/Time(PDGEMM)					
Machine	Procs	Block Size	N = 2000	N = 4000	N = 10000
Cray T3E	4	32	.50	.40	
	16		.75	.51	.40
	64		1.86	.72	.45
IBM SP2	4	50	.60		
	16		1.16	.64	
	64		2.24	1.03	.51
Intel XP/S GP Paragon	4	32	.52		
	16		.89	.50	
	64		2.08	.79	.44
Berkeley NOW	4	32	.44		
	32		.88	.54	.47
	64		1.18	.62	.49

Out of "Core" Algorithms

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 ScaLAPACK matrices

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

QR Factorization on 64 processors Intel Paragon

Out-of-core / In-core

Source: Jack Dongarra

Recursive Algorithms

- Still uses delayed updates, but organized differently
  - (formulas on board)
- Can exploit recursive data layouts
  - 3x speedups on least squares for tall, thin matrices
- Theoretically optimal memory hierarchy performance
- See references at
  - http://lawra.uni-c.dk/lawra/index.html
  - http://www.cs.umu.se/research/parallel/recursion/

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

LU Algorithm:

1: Split matrix into two rectangles (m x 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

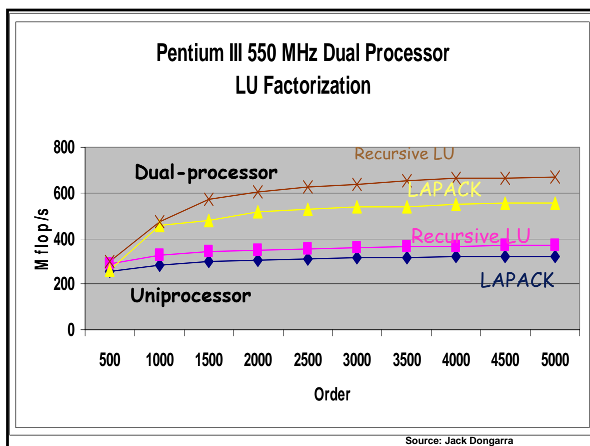
L, A_{12}, A_{21}, A_{22}

Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, …

Source: Jack Dongarra

Recursive Factorizations

- Just as accurate as conventional method
- Same number of operations
- Automatic variable-size 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).



Source: Jack Dongarra

Recursive Algorithms – Limits

- Two kinds of dense matrix compositions
- One Sided
  - Sequence of simple operations applied on left of matrix
  - Gaussian Elimination: A = L*U or A = P*L*U
    - Symmetric Gaussian Elimination: $A = L*D*L^T$
    - Cholesky: $A = L*L^T$
  - QR Decomposition for Least Squares: A = Q*R
  - Can be nearly 100% BLAS 3
  - Susceptible to recursive algorithms
- Two Sided
  - Sequence of simple operations applied on both sides, alternating
  - Eigenvalue algorithms, SVD
  - At least ~25% BLAS 2
  - Seem impervious to recursive approach?
  - Some recent progress on SVD (25% vs 50% BLAS2)

LAPACK and ScaLAPACK Status

- "One-sided Problems" are scalable
  - In Gaussian elimination, A factored into product of 2 matrices A = LU by premultiplying A by sequence of simpler matrices
  - Asymptotically 100% BLAS3
  - LU ("Linpack Benchmark")
  - Cholesky, QR
- "Two-sided Problems" are harder
  - A factored into product of 3 matrices by pre and post multiplication
  - Half BLAS2, not all BLAS3
  - Eigenproblems, SVD
    - Nonsymmetric eigenproblem hardest
- Narrow band problems hardest (to do BLAS3 or parallelize)
  - Solving and eigenproblems
- www.netlib.org/{lapack,scalapack}

Some contributors (incomplete list)

Participants

Krste Asanovic (UC Berkeley)
Richard Barrett (U. Tenn)
Jeff Bilmes (UC Berkeley)
Susan Blackford (ORNL)
Tony Chan (UCLA)
Jaeyoung Choi (LBNL)
Ed D'Azevedo (ORNL)
Inderjit Dhillon (UC Berkeley)
Jack Dongarra (U Tenn, ORNL)
Jeremy Du Croz (NAG)
Stan Eisenstat (Yale)
John Gilbert (Xerox PARC)
Sven Hammarling (NAG)
Greg Henry (Intel)
Steve Huss-Lederman (SRC)
W. Kahan (UC Berkeley)
Rencang Li (UC Berkeley)
Joseph Liu (York)
Antoine Petitet (U Tenn)
Roldan Pozo (U Tenn)
Huan Ren (UC Berkeley)
Charles Romine (ORNL)
Ivan Slapničar (U Split)
Ken Stanley (UC Berkeley)
Bernard Tourancheau (U Tenn)
Robert van de Geijn (U Texas)
Paul Van Dooren (U Illinois)
David Walker (ORNL)
Kathy Yelick (UC Berkeley)

Zhaojun Bai (U Kentucky)
Michael Berry (U Tenn)
Chris Bischof (ANL)
Soumen Chakrabarti (UC Berkeley)
Chee-Whye Chin (UC Berkeley)
Andy Cleary (LLNL)
Jim Demmel (UC Berkeley)
June Donato (ORNL)
Zlatko Drmač (U Hagen)
Victor Eijkhout (UCLA)
Vince Fernando (NAG)
Ming Gu (UC Berkeley, LBL)
Mike Heath (U Illinois)
Dominic Lam (UC Berkeley)
Bo Kågström (U Umeå)
Youngbae Kim (U Tenn)
Xiaoye Li (UC Berkeley)
Beresford Parlett (UC Berkeley)
Peter Poromaa (U Umeå)
Padma Raghavan (U Illinois)
Howard Robinson (UC Berkeley)
Jeff Rutter (UC Berkeley)
Dan Sorensen (Rice U)
Xiaobai Sun (ANL)
Anna Tsao (SRC)
Henk van der Vorst (Utrecht U)
Krešimir Veselić (U Hagen)
Clint Whaley (U Tenn)

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

Supported by ARPA, NSF, DOE