The Impact of RISC

and Parallel RISC Systems

on Linear Algebra Software

Jack Dongarra

Computer Science Department
University of Tennessee

and

Mathematical Sciences Section
Oak Ridge National Laboratory

(http://www.netlib.org/utk/people/JackDongarra.html)

Overview

- Motivation - History Level 1 BLAS
- Linpack
- Unrolling to allow compiler to reconize
- Why
- Tricks like the babe algorithm
- Supercomputers
- Vector Computers
- Level 2 3 BLAS
- Compilers
- Memory Heirarchy
- RISC
- Linpack on RISC
- MP
- Scalapack
- Bandwidth Latency
- Iterative Methods (Bombardment)
- Future

Mathematical Software for Linear Algebra in the 1970's

- 1970's software packages for eigenvalue problems and linear equations.
- EISPACK - Fortran routines
  - Eigenvalue problem
  - Wilkinson - Reinsch Handbook for Automatic Computation
- LINPACK - Fortran routines
  - Systems of linear equations.
  - Collective ideas.

Mathematical Software for Linear Algebra in the 1970's

- EISPACK early 70's
- Fortran translation of Algol algorithms from Num. Math.
- Software issues -
  - Portability
  - Robustness
  - Accuracy
  - Uniform
  - Well documented
- Little thought given to high performance computers.
- CDC 7600 state-of-the-art supercomputer.

Mathematical Software for Linear Algebra in the 1970's

- Level 1 BLAS
- Basic Linear Algebra Subprograms for Fortran Usage
- C. Lawson, R. Hanson, D. Kincaid, and F. Krogh
- Conceptial aid in design and coding
- Aid to readability and documentation
- Promote efficieny:
  - through optimization or assembly language versions
- Improve robustness and reliability
- Improve portability through standardization

Mathematical Software for Linear Algebra in the 1970's

Level 1 BLAS

- Dot products
- 'axpy' operations $y \leftarrow \alpha x + y$
- Multiple a vector by a constant
- Set up and apply Givens rotations
- Copy and swap vectors
- Vector norms

	name	dim	scalars	vector	vector	scalars
SUBROUTINE	_AXPY	(N,	ALPHA	X, INCX,	Y, INCY)	
FUNCTION	_DOT_	(N,		X, INCX,	Y, INCY)	
SUBROUTINE	_SCAL	(N,	ALPHA	X, INCX)		
SUBROUTINE	_ROTG	(	A, B			C,S)
SUBROUTINE	_ROT	(N		X, INCX,	Y, INCY	C,S)
SUBROUTINE	_COPY	(N,		X, INCX,	Y, INCY)	
SUBROUTINE	_SWAP	(N,		X, INCX,	Y, INCY)	
FUNCTION	_NRM2	(N,		X, INCX)		
FUNCTION	_ASUM	(N,		X, INCX)		
FUNCTION	_ASUM	(N,		X, INCX)		

Mathematical Software for Linear Algebra in the 1970's

- LINPACK late 70's
- Used current ideas - not a translation.
- First vector supercomputer arrived - CRAY 1.
- BLAS to standardize basic vector operations.
- LINPACK embraced BLAS for modularity and efficiency.
- Reworked algorithms.
- Column oriented.

Here is the body of the code of the LINPACK routine SPOFA, which implements the above 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
```

BLAS – BASICS

Level 1 BLAS – Vector Operations (Late '70s)

$$\begin{array}{lll} y \leftarrow y + \alpha x, & y \leftarrow x, & y \leftarrow \alpha x, \\ \alpha \leftarrow x^T y, & \alpha \leftarrow \|x\|_2, & y \leftrightarrow x. \end{array}$$

Example: `SAXPY` operate with vectors:

$$y \leftarrow y + \alpha x$$

- 2 vector loads
- 2 vector operations
- 1 vector store

Too much memory traffic: little chance to use the memory hierarchy,

- $O(n)$ memory references,
- $O(n)$ floating point operations.

Loop Unrolling

In scalar case

```
do i = 1,n,4   y(1:n) = y(1:n) + alfa*x(1:n)
   y(i) = y(i) + alfa*x(i)
   y(i+1) = y(i+1) + alfa*x(i+1)
   y(i+2) = y(i+2) + alfa*x(i+2)
   y(i+3) = y(i+3) + alfa*x(i+3)
continue
```

- WHY?
  - Reduce the loop overhead
  - Allow pipelined functional units to operate
  - Allow independent functional units to operate
  - Vector instructions
- Operation to data reference
  - $2n$ operations
  - $3n$ data transfer
- Machines that might benefit?
  - Thought would work well on all scalar machines, but... vector machines

Other Tricks for Performance...

Algorithm for factoring a tridiagonal matrix

$$\begin{pmatrix} x & x & & & & & & & \\ x & x & x & & & & & & \\ & x & x & x & & & & & \\ & & x & x & x & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & \ddots & \ddots & \ddots & \\ & & & & & & x & x & x \\ & & & & & & & x & x & x \\ & & & & & & & & x & x \end{pmatrix}$$

- Algorithm is sequential in nature.
- Not much scope for pipelining.
- In an attempt to increase the speed of execution the BABE algorithm was adopted.

Burn At Both Ends (BABE)

With the BABE algorithm the factoerization is started at the top of the matrix as well as the bottom of the matrix at the same time.

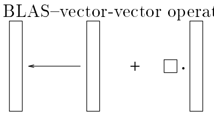
$$\begin{pmatrix} x & x & & & & & & & & \\ 0 & x & x & & & & & & & \\ & x & x & x & & & & & & \\ & & x & x & x & & & & & \\ & & & \ddots & \ddots & \ddots & & & & \\ & & & & \ddots & \ddots & \ddots & & & \\ & & & & & \ddots & \ddots & \ddots & & \\ & & & & & & x & x & x & \\ & & & & & & & x & x & x \\ & & & & & & & & x & x & 0 \\ & & & & & & & & & x & x \end{pmatrix} \quad \begin{pmatrix} x & x & & & & & & & & \\ 0 & x & x & & & & & & & \\ & 0 & x & x & & & & & & \\ & & x & x & x & & & & & \\ & & & \ddots & \ddots & \ddots & & & & \\ & & & & \ddots & \ddots & \ddots & & & \\ & & & & & \ddots & \ddots & \ddots & & \\ & & & & & & x & x & x & \\ & & & & & & & x & x & 0 \\ & & & & & & & & x & x & 0 \\ & & & & & & & & & x & x \end{pmatrix}$$

$$\begin{pmatrix} x & x & & & & & & & & & \\ 0 & x & x & & & & & & & & \\ & 0 & x & x & & & & & & & \\ & & \ddots & \ddots & \ddots & & & & & & \\ & & & \ddots & \ddots & \ddots & & & & & \\ & & & & \ddots & \ddots & \ddots & & & & \\ & & & & & 0 & x & x & & & \\ & & & & & & 0 & x & 0 & & \\ & & & & & & & x & x & 0 & \\ & & & & & & & \ddots & \ddots & \ddots & \\ & & & & & & & & \ddots & \ddots & \ddots \\ & & & & & & & & & \ddots & \ddots & \ddots \\ & & & & & & & & & & x & x & 0 \\ & & & & & & & & & & & x & x \end{pmatrix}$$

- Reduce loop overhead by a factor of 2
- Scope for independent operations
- Resulted in 25% faster execution

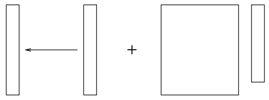
Level 1, 2, and 3 BLAS

- Level 1 BLAS–vector-vector operations



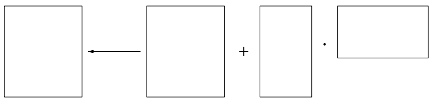
$y \leftarrow y + \alpha x$, also dot product, ...

- Level 2 BLAS–matrix-vector operations



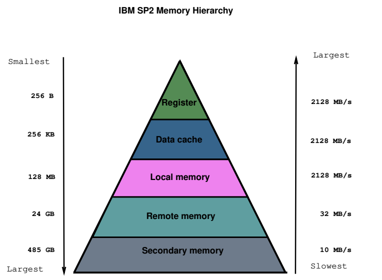
$y \leftarrow y + Ax$, also triangular solve, rank-1 update

- Level 3 BLAS–matrix-matrix operations



$A \leftarrow A + BC$, also block triangular solve, rank-k update

Why Higher Level BLAS?



- Can only do arithmetic on data at top
  $\longrightarrow$ keep active data as close to top of hierarchy as possible
- Higher level BLAS lets us do this:

		mem ref	flops	flops/mem ref
Level 1 BLAS	$y \leftarrow y + \alpha x$	3n	2n	2/3
Level 2 BLAS	$y \leftarrow y + Ax$	n^2	$2n^2$	2
Level 3 BLAS	$A \leftarrow A + BC$	$4n^2$	$2n^3$	n/2

- On parallel machines
  higher level BLAS $\longrightarrow$increase granularity $\longrightarrow$ lower synchronization cost

Matrix-vector product

DOT version-25 Mflops in cache
(Model 530, 50 Mflop/s peak)

```
     DO 20 I = 1, M
        DO 10 J = 1, N
           Y(I) = Y(I) + A(I,J)*X(J)
  10    CONTINUE
  20 CONTINUE
```

From	Cache	22.7 Mflops
	Memory	12.4 Mflops

Loop unrolling

```
     DO 20 I = 1, M, 2
        T1 = Y(I  )
        T2 = Y(I+1)
        DO 10 J = 1, N
           T1 = T1 + A(I,J  )*X(J)
           T2 = T2 + A(I+1,J)*X(J)
  10    CONTINUE
        Y(I  ) = T1
        Y(I+1) = T2
  20 CONTINUE
```

3 loads, 4 flops

Speed of $y \leftarrow y + A^T x, N = 48$

Model 530 (50 Mflop/s peak)

Depth	1	2	3	4	∞
Speed	25	33.3	37.5	40	50
Measured	22.7	30.5	34.3	36.5	-
(Memory)	12.4	12.7	12.7	12.6	-

Matrix-matrix multiply

DOT version - 25 Mflops in cache

```
   DO 30 J = 1, M
      DO 20 I = 1, M
         DO 10 K = 1, L
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
10       CONTINUE
20    CONTINUE
30 CONTINUE
```


How to get 50 Mflops!

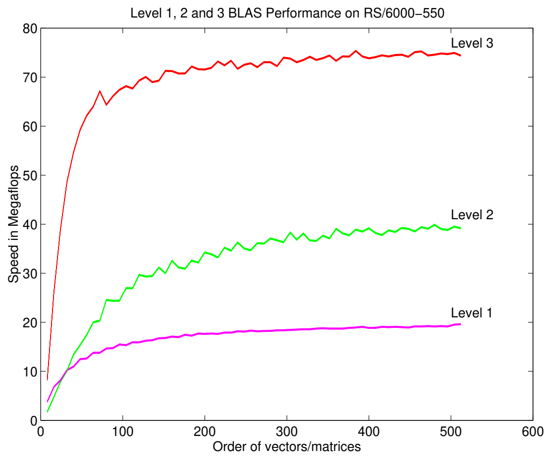
```
   DO 30 J = 1, M, 2
      DO 20 I + 1, M, 2
         T11 = C(I,   J  )
         T12 = C(I,   J+1)
         T21 = C(I+1,J  )
         T22 = C(I+1,J+1)
         DO 10 K = 1, L
            T11 = T11 + A(I,  K)*B(K,J  )
            T12 = T12 + A(I,  K)*B(K,J+1)
            T21 = T21 + A(I+1,K)*B(K,J  )
            T22 = T22 + A(I+1,K)*B(K,J+1)
10       CONTINUE
         C(I,   J  ) = T11
         C(I,   J+1) = T12
         C(I+1,J  ) = T21
         C(I+1,J+1) = T22
20    CONTINUE
30 CONTINUE
```

Inner loop: <u>4 loads</u>, 8 operations, optimal.

In practice we have measured 48.1

(M=N=16, L=128)

BLAS – BASICS



- Development of blocked algorithms using Level 3 BLAS,
- ⇒ LAPACK, high performance, portability.

BLAS – REFERENCES

- BLAS software and documentation can be obtained via:
  - WWW: `http://www.netlib.org/blas`,
  - (anonymous) ftp `netlib2.cs.utk.edu`:

    `cd blas; get index`
  - email `netlib@ornl.gov` with the message:

    `send index from blas`
- Comments and questions can be addressed at

  `lapack@cs.utk.edu`

- C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, 5:308–325, 1979.
- J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14(1):1–32, (1988).
- J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16(1):1–17, 1990.

Block Algorithms and their Derivation

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ . & a_{jj} & \alpha_j^T \\ . & . & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}$$

and equating coefficients of the j^{th} column, we obtain:

$$\begin{aligned} a_j &= U_{11}^T u_j \\ a_{jj} &= u_j^T u_j + u_{jj}^2. \end{aligned}$$

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

$$\begin{aligned} U_{11}^T u_j &= a_j \\ u_{jj}^2 &= a_{jj} - u_j^T u_j. \end{aligned}$$

Here is the body of the code of the LINPACK routine SPOFA, which implements the above 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
```



```
   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
```

- change by itself is sufficient to make big gains in performance on a number of machines
- from 24 to 34 megaflops for a matrix of order 500 on an IBM RS/6000 model 550.
- from 72 to 251 megaflops for a matrix of order 500 on one processor of a CRAY Y-MP

To derive a block form of Cholesky factorization, we write the defining equation in partitioned form thus:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ . & A_{22} & A_{23} \\ . & . & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ U_{12}^T & U_{22}^T & 0 \\ U_{13}^T & U_{23}^T & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{pmatrix}.$$

Equating submatrices in the second block of columns, we obtain:

$$\begin{aligned} A_{12} &= U_{11}^T U_{12} \\ A_{22} &= U_{12}^T U_{12} + U_{22}^T U_{22}. \end{aligned}$$

Hence, if U_{11} has already been computed, we can compute U_{12} as the solution to the equation

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

by a call to the Level 3 BLAS routine STRSM; and then we can compute U_{22} from

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

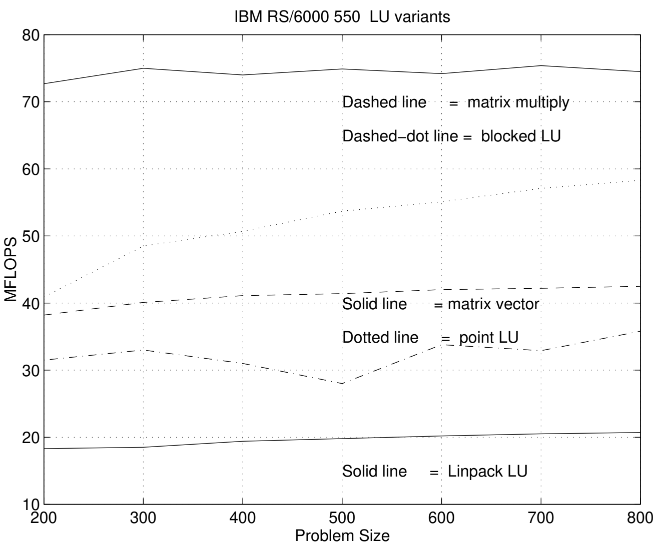
Speed in megaflops of Cholesky factorization $A = U^T U$ for $n = 500$

Machine:	RS/6000 - 500
j-variant: LINPACK	24
j-variant: using Level 2 BLAS	34
j-variant: using Level 3 BLAS	69

```
   DO 10 J = 1, N, NB
      JB = MIN( NB, N-J+1 )
      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
```


Blocked Algorithms have been developed for:

- LU factorization
- Cholesky factorization
- Factorization of symmetric indefinite matrices
- Matrix inversion
- Banded LU and Cholesky factorization
- QR factorization
- Form Q or $Q^T B$
- Orthogonal reduction to:
  - Hessenberg
  - symmetric tridiagonal
  - bidiagonal
- Block QR iteration for nonsymmetric eigenvalue problems



Performance Numbers Comparing Various RISC Processors

Using the Linpack Benchmark

(All based on actual runs.)
Linpack n=100 based on Fortran code only.

Ax=b n=1000 is based on a blocked algorithm (LAPACK routine) using the Level 3 BLAS as provided by the vendor.

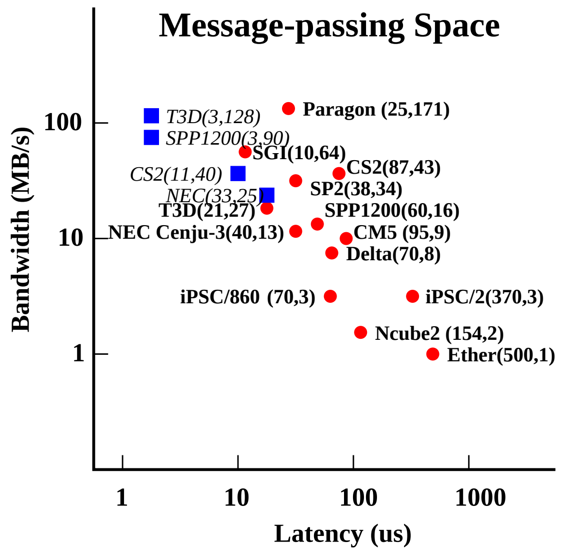
Machine	MHz	Cycle time nsec	Linpack n=100 Mflops	% peak	Ax = b n=1000 Mflops	% peak	Theor Peak Mflops
DEC Alpha	300	3.3	140	23%	411	69%	600
IBM Power2	66	15	130	48%	236	89%	266
SGI Power	75	13.3	101	33%	260	87%	300
HP 735	100	10	61	31%	107	54%	198
DEC Alpha	200	5	43	22%	155	78%	200
DEC Alpha	182	5	39	21%	141	77%	182
IBM RS-580	66	15	38	30%	104	83%	125
DEC Alpha	160	6	36	23%	114	71%	160
DEC Alpha	150	7	30	20%	107	71%	150
IBM RS-550	42	24	26	31%	70	83%	84
HP 730/750	66	15	24	36%	47	71%	66
SGI Indy2/Crim	100	10	15	30%	32	64%	50
IBM RS-530	25	40	15	30%	42	84%	50
KSR (1 proc)	40	25	15	38%	31	78%	40
Intel i860	40	25	10	25%	34	85%	40
CRAY T90	454	2.2	522	29%	1576	88%	1800
CRAY C90	238	4.2	387	41%	902	95%	952
CRAY J90	100	10	115	51%	193	96%	200
CRAY Y-MP	166	6	161	48%	324	97%	333
CRAY X-MP	118	8.5	121	51%	218	93%	235
CRAY 3	481	2.1	241	25%			962
CRAY 2	244	4.1	120	25%	384	79%	488
CRAY 1	80	12.5	27	17%	110	69%	160

Table 1: Multiprocessor Latency and Bandwidth.

Machine	OS	Latency $n = 0$ (μs)	Bandwidth $n = 10^6$ (MB/s)	$n_{1/2}$ bytes	Theoretical Bandwidth (MB/s)
Convex SPP1200 (PVM)	SPP-UX 3.0.4.1	63	15	1000	250
Convex SPP1200 (sm m-n)	SPP-UX 3.0.4.1	11	71	1000	250
Cray T3D (sm)	MAX 1.2.0.2	3	128	363	300
Cray T3D (PVM)	MAX 1.2.0.2	21	27	1502	300
Intel Paragon	OSF 1.0.4	29	154	7236	175
Intel Paragon	SUNMOS 1.6.2	25	171	5856	175
Intel Delta	NX 3.3.10	77	8	900	22
Intel iPSC/860	NX 3.3.2	65	3	340	3
Intel iPSC/2	NX 3.3.2	370	2.8	1742	3
IBM SP-1	MPL	270	7	1904	40
IBM SP-2	MPI	35	35	3263	40
KSR-1	OSF R1.2.2	73	8	635	32
Meiko CS2 (sm)	Solaris 2.3	11	40	285	50
Meiko CS2	Solaris 2.3	83	43	3559	50
nCUBE 2	Vertex 2.0	154	1.7	333	2.5
nCUBE 1	Vertex 2.3	384	0.4	148	1
NEC Cenju-3	Env. Rel 1.5d	40	13	900	40
NEC Cenju-3 (sm)	Env. Rel 1.5d	34	25	400	40
SGI	IRIX 6.1	10	64	799	1200
TMC CM-5	CMMD 2.0	95	9	962	10
Ethernet	TCP/IP	500	0.9		1.2
FDDI	TCP/IP	900	9.7		12
ATM-100	TCP/IP	900	3.5		12

Table 2: Computation Performance.

Machine	OS	Clock cycle MHz	Clock cycle (nsec)	Linpack 100 Mfls	Linpack 100 (ops/cl)	Linpack 1000 Mfls	Linpack 1000 (ops/cl)	Latency us	Latency (cl)
Convex SPP1200 (PVM)	SPP-UX 3.0.4.1	100	(8.33)	65	(.54)	123	(1.02)	63	(7560)
Convex SPP1200 (sm m-n)								11	(1260)
Cray T3D (sm)	MAX 1.2.0.2	150	(6.67)	38	(.25)	94	(.62)	3	(450)
Cray T3D (PVM)								21	(3150)
Intel Paragon	OSF 1.0.4	50	(20)	10	(.20)	34	(.68)	29	(1450)
Intel Paragon	SUNMOS 1.6.2							25	(1250)
Intel Delta	NX 3.3.10	40	(25)	9.8	(.25)	34	(.85)	77	(3080)
Intel iPSC/860	NX 3.3.2	40	(25)	9.8	(.25)	34	(.85)	65	(2600)
Intel iPSC/2	NX 3.3.2	16	(63)	.37	(.01)	–	(–)	370	(5920)
IBM SP-1	MPL	62.5	(16)	38	(.61)	104	(1.66)	270	(16875)
IBM SP-2	MPI	66	(15.15)	130	(1.97)	236	(3.58)	35	(2310)
KSR-1	OSF R1.2.2	40	(25)	15	(.38)	31	(.78)	73	(2920)
Meiko CS2 (MPI)	Solaris 2.3	90	(11.11)	24	(.27)	97	(1.08)	83	(7470)
Meiko CS2 (sm)								11	(990)
nCUBE 2	Vertex 2.0	20	(50)	.78	(.04)	2	(.10)	154	(3080)
nCUBE 1	Vertex 2.3	8	(125)	.10	(.01)	–	(–)	384	(3072)
NEC Cenju-3	Env Rev 1.5d	75	(13.3)	23	(.31)	39	(.52)	40	(3000)
NEC Cenju-3(sm)	Env Rev 1.5d	75	(13.3)	23	(.31)	39	(.52)	34	(2550)
SGI Power Challenge	IRIX 6.1	90	(11.11)	126	(1.4)	308	(3.42)	10	(900)
TMC CM-5	CMMD 2.0	32	(31.25)	–	(–)	–	(–)	95	(3040)



Challenges in developing distributed memory libraries

- How to integrate libraries?
  - no standard software
  - many parallel languages
  - many flavors of message passing
  - various parallel programming models
  - assumptions made about parallel environment
    - granularity
    - topology
    - overlapping of communication / computation
    - development tools
- Where is the data?
  - who owns it?
  - optimal data distribution not a function of individual routines but of overall integration of components
- Who determines data layout and/or transformations?
  - determined by user?
  - determined by library developer?
  - choose from a small set?
  - allow for dynamic data distributions?
  - load balancing?

PBLAS – INTRODUCTION

Parallel Basic Linear Algebra Subprograms for distributed memory MIMD computers.

- **Similar functionality** as the BLAS: distributed vector-vector, matrix-vector and matrix-matrix operations,
- **Simplification of the parallelization** of dense linear algebra codes: especially when implemented on top of the BLAS,
- **Clarity:** code is shorter and easier to read,
- **Modularity:** gives programmer larger building blocks,
- **Program portability:** machine dependency are confined to the PBLAS (BLAS and BLACS).

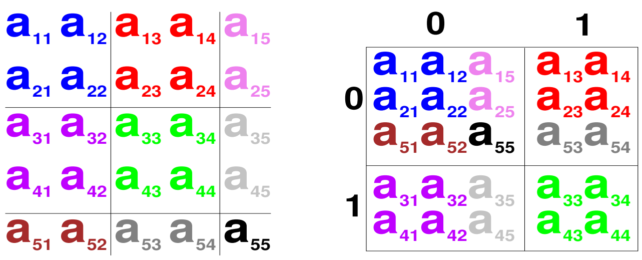
PBLAS – STORAGE CONVENTIONS

- An `M_`-by-`N_` **matrix is block partitionned** and these `MB_`-by-`NB_` blocks are distributed according to the

  2-dimensional block-cyclic scheme
  ⇒ **load balanced computations, scalability**,

- Locally, the scattered **columns are stored contiguously** (FORTRAN "Column-major")

  ⇒ **re-use of the BLAS** (leading dimension `LLD_`).



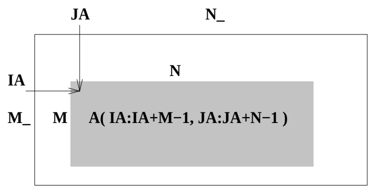
5 x 5 matrix partitioned in 2 x 2 blocks   2 x 2 process grid point of view

Descriptor `DESC_`: 8-Integer array describing the matrix layout, containing `M_`, `N_`, `MB_`, `NB_`, `RSRC_`, `CSRC_`, `CTXT_` and `LLD_`, where (`RSRC_`, `CSRC_`) are the coordinates of the process owning the first matrix entry in the grid specified by `CTXT_`.

Ex: `M_=N_=5`, `MB_=NB_=5`, `RSRC_=CSRC_= 0`, `LLD_` $\geq$ `3` (in process row 0), and `2` (in process row 1).

PBLAS – ARGUMENT CONVENTIONS

- **Global view of the matrix operands**, allowing global addressing of distributed matrices (hiding complex local indexing),



- **Code reusability**, interface very close to sequential BLAS:

```
 CALL  DGEXXX( M, N, A( IA, JA ), LDA )

 CALL  DGEMM( 'No Transpose', 'No Transpose',
$     M-J-JB+1, N-J-JB+1, JB, -ONE, A(J+JB,J),
$     LDA, A(J,J+JB), LDA, ONE, A(J+JB,J+JB),
$     LDA )
```

↓

```
 CALL PDGEXXX( M, N, A, IA, JA, DESCA )

 CALL PDGEMM( 'No Transpose', 'No Transpose',
$     M-J-JB+1, N-J-JB+1, JB, -ONE, A, J+JB,
$     J, DESCA, A, J, J+JB, DESCA, ONE, A,
$     J+JB, J+JB, DESCA )
```


PBLAS – SPECIFICATIONS

```
DGEMV( TRANS, M, N, ALPHA, A, LDA, X, INCX,
       BETA, Y, INCY )
```

↕

```
PDGEMV( TRANS, M, N, ALPHA, A, IA, JA, DESCA,
        X, IX, JX, DESCX, INCX,
        BETA, Y, IY, JY, DESCY, INCY )
```

- In the PBLAS, the increment specified for vectors is always global. So far only `INCX=1` and `INCX=DESCX(1)` are supported.

BLAS	PBLAS
`INTEGER LDA`	`INTEGER IA, JA, DESCA( 8 )`
`INTEGER INCX`	`INTEGER INCX, IX, JX, DESCX( 8 )`
`A, LDA`	`A, IA, JA, DESCA`
`X, INCX`	`X, IX, JX, DESCX, INCX`

- PBLAS matrix transposition routine (REAL):

```
PDTRAN( M, N, ALPHA, A, IA, JA, DESCA, BETA,
        C, IC, JC, DESCC )
```

- Level 1 BLAS functions have become PBLAS subroutines. Ouput Scalar correct in operand scope.

>> **SPMD PROGRAMMING MODEL** <<

Parallel Level 2 and 3 BLAS: PBLAS

Goal

- Port sequential library for shared memory machine that use the BLAS to distributed memory machines with little effort.
- Reuse the existing software by hiding the message passing in a set of BLAS routines.
- Parallel implementation of the BLAS that understands how the matrix is layed out and when called can perform not only the operation but the required data transfer.

LAPACK → ScaLAPACK

BLAS → PBLAS (BLAS, BLACS)

- Quality (maintenance)
- Portability (F77 - C)
- Efficiency - Reuseability (BLAS, BLACS)
- Hide Parallelism in (P)BLAS

SEQUENTIAL LU FACTORIZATION CODE

```
 DO 20 J = 1, MIN( M, N ), NB
    JB = MIN( MIN( M, N )-J+1, NB )


    Factor diagonal and subdiagonal blocks and test for exact
    singularity.

    CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ),
$                IINFO )

    Adjust INFO and the pivot indices.

    IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
    DO 10 I = J, MIN( M, J+JB-1 )
       IPIV( I ) = J - 1 + IPIV( I )
10  CONTINUE

    Apply interchanges to columns 1:J-1.

    CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )


    IF( J+JB.LE.N ) THEN

       Apply interchanges to columns J+JB:N.

       CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1,
$                   IPIV, 1 )

       Compute block row of U.

       CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
$                  JB, N-J-JB+1, ONE, A( J, J ), LDA,
$                  A( J, J+JB ), LDA )
       IF( J+JB.LE.M ) THEN

          Update trailing submatrix.

          CALL DGEMM( 'No transpose', 'No transpose',
$                     M-J-JB+1, N-J-JB+1, JB, -ONE,
$                     A( J+JB, J ), LDA, A( J, J+JB ), LDA,
$                     ONE, A( J+JB, J+JB ), LDA )
       END IF
    END IF
20 CONTINUE
```

PARALLEL LU FACTORIZATION CODE

```
 DO 10 J = JA, JA+MIN(M,N)-1, DESCA( 4 )
    JB = MIN( MIN(M,N)-J+JA, DESCA( 4 ) )
    I = IA + J - JA

    Factor diagonal and subdiagonal blocks and test for exact
    singularity.

    CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )


    Adjust INFO and the pivot indices.

    IF( INFO.EQ.0 .AND. IINFO.GT.0 )
$      INFO = IINFO + J - JA



    Apply interchanges to columns JA:J-JA.

    CALL PDLASWP( 'Forward', 'Rows', J-JA, A, IA, JA, DESCA,
$                 J, J+JB-1, IPIV )

    IF( J-JA+JB+1.LE.N ) THEN

       Apply interchanges to columns J+JB:JA+N-1.

       CALL PDLASWP( 'Forward', 'Rows', N-J-JB+JA, A, IA,
$                    J+JB, DESCA, J, J+JB-1, IPIV )

       Compute block row of U.

       CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
$                   JB, N-J-JB+JA, ONE, A, I, J, DESCA, A, I,
$                   J+JB, DESCA )
       IF( J-JA+JB+1.LE.M ) THEN

          Update trailing submatrix.

          CALL PDGEMM( 'No transpose', 'No transpose',
$                      M-J-JB+JA, N-J-JB+JA, JB, -ONE, A,
$                      I+JB, J, DESCA, A, I, J+JB, DESCA,
$                      ONE, A, I+JB, J+JB, DESCA )
       END IF
    END IF
10 CONTINUE
```


SCALAPACK – ONGOING WORK

- **HPF**
  - HPF supports the 2D-block-cyclic distribution used by ScaLAPACK
  - HPF like calling sequence (Global indexing scheme)
- **Portability: MPI implementation of the BLACS**
- **More flexibility added to the PBLAS**
- **More testing and timing programs**
- **Condition estimators**
- **Iterative refinement of linear system solutions**
- **SVD**
- **Linear Least Square solvers**
- **Banded systems**
- **Non symmetric eigensolvers**
- ...

SCALAPACK – REFERENCES

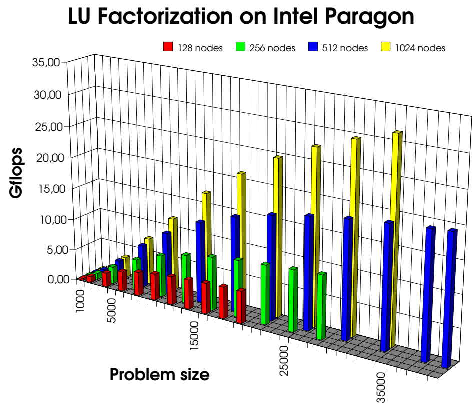
- ScaLAPACK software and documentation can be obtained via:
  - WWW: `http://www.netlib.org/scalapack`,
  - WWW: `http://www.netlib.org/lapack/lawns`,
  - (anonymous) ftp `netlib2.cs.utk.edu`:
    - cd scalapack; get index
    - cd lapack/lawns; get index
  - email `netlib@ornl.gov` with the message:

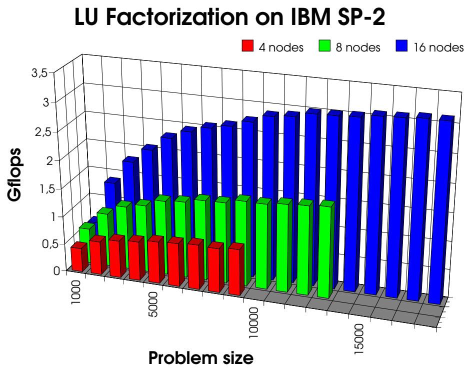
    `send index from scalapack`
- Comments and questions can be addressed at

  `scalapack@cs.utk.edu`
- LAPACK Working Notes:

  #43, #55, #57, #58, #61, #65, #73, #80, #86, #91,

  #92, **#93**, **#94**, **#95**, #96, **#100**.
- J. Dongarra and D. Walker, *Software Libraries for Linear Algebra Computations on High Performance Computers*, SIAM Review, Vol. 37, (2), pp. 151 – 180, 1995.





Divide

- Important on Tridiagonal Solver for ADI Schemes.
- $9n$ floating point operations in the inner loop, n are floating-point divide instructions.
- Also important on block tridiagonal and pentadiagonal solvers.
- Newton's iteration usually used
- Not typically pieplined

Processor	Clock (Mhz)	Divide (CP)	Cache BW (MW/s)	Memory BW (MW/s)	Peak Perf. (MF/s)	Linpack (MF/s)
Cray C-90/1	240	4	-	1440	960	387
Cray Y-MP/1	166	4	-	500	333	161
RS/6000-590	66	19	266	266	264	130
DEC Alpha	150	63	150	37	150	30
HP PA-RISC	99	20	99	33	198	41
Intel i860	40	190	80	16	60	10
SGI (R4400)	100	36	100	50	50	17
Super SPARC	40	7	40	40	40	7

Sparse Matrix Algorithms

- Based on iterative methods
- Important for many large applications
  - Krylov subspace methods
  - Multigrid
- Attempts to solve $Ax = b$, where A is large and sparse.
  - A too large for cache
  - Generates a sequence of iterates, x_i converging to solution.
  - Requires accessing A for each iteration.
  - Thus, computation runs at main memory speeds, not cache speeds.
- Method may not converge.

Conclusion

- RISC has had an impact on basic software design
- Exploiting memory heirarchy in the algorithm
- Blocking is now taken for granted