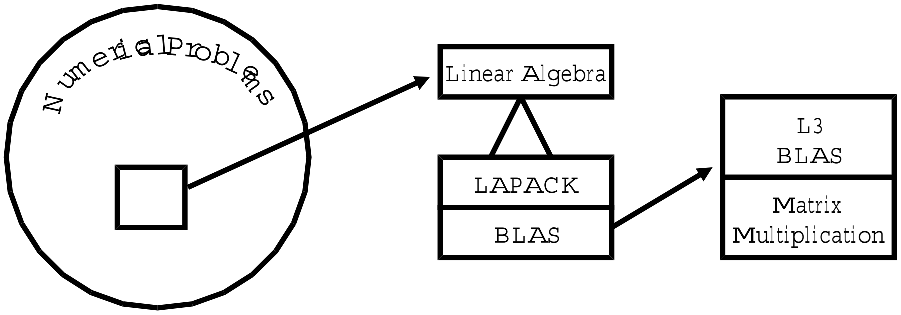
6 LAPACK

So far, we have seen ideas of:

- Cluster architecture
- Sequential and parallel computing performance evaluation
- Practice on sequential computing with matrix multiplication
- Cluster programming with MPI
- Communication evaluation on clusters
- *Rough* analysis of granularity

However, remind we started with



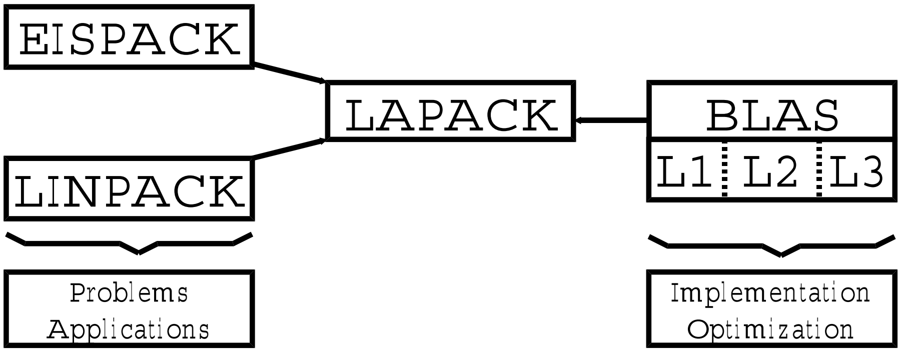
From LAPACK Homepage http://www.netlib.org/lapack/

"LAPACK is written in Fortran77 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision."

"The original goal of the LAPACK project was to make the widely used EISPACK and LINPACK libraries run efficiently on shared-memory vector and parallel processors."

"LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). While LINPACK and EISPACK are based on the vector operations kernels of the Level 1 BLAS, LAPACK was designed at the outset to exploit the L3 BLAS"

Acknowledgements: ... NSF Grant ... DOE Grant ...



From LINPACK Homepage http://www.netlib.org/linpack/

"**LINPACK** is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems."

From EISPACK Homepage http://www.netlib.org/eispack/

"**EISPACK** is a collection of Fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices: complex general, complex Hermitian, real general, real symmetric, real symmetric banded, real symmetric tridiagonal, special real tridiagonal, generalized real, and generalized real symmetric matices. In addition, two routines are included that use singular value decomposition to solve certain least-squares problems."

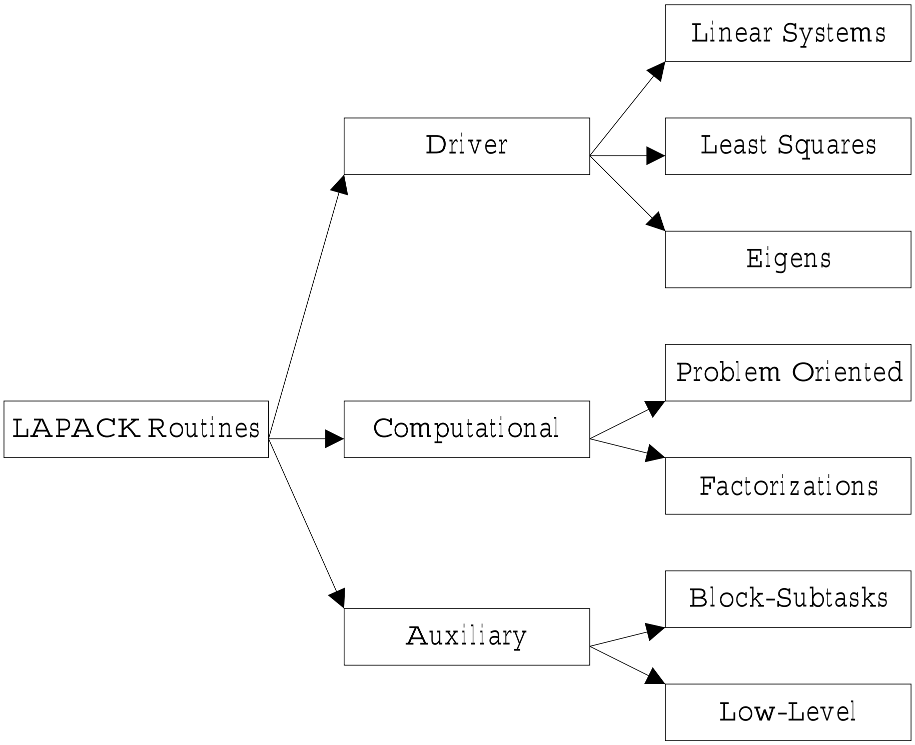
Finally, there are a lot of functions (well, subroutines...), each one with a lot of parameters. Classifications (from LAPACK documentation):

- driver routines, each of which solves a complete problem, for example solving a system of linear equations, or computing the eigenvalues of a real symmetric matrix. Users are recommended to use a driver routine if there is one that meets their requirements.

- computational routines, each of which performs a distinct computational task, for example an LU factorization, or the reduction of a real symmetric matrix to tridiagonal form. Each driver routine calls a sequence of computational routines. Users (especially software developers) may need to call computational routines directly to perform tasks, or sequences of tasks, that cannot conveniently be performed by the driver routines.

- Auxiliary routines, which in turn can be classified as follows:
  - routines that perform subtasks of block algorithms – in particular, routines that implement unblocked versions of the algorithms;
  - routines that perform some commonly required low-level computations, for example scaling a matrix, computing a matrix-norm, or generating an elementary Householder matrix; some of these may be of interest to numerical analysts or software developers and could be considered for future additions to the BLAS;
  - a few extensions to the BLAS, such as routines for applying complex plane rotations, or matrix-vector operations involving complex symmetric matrices (the BLAS themselves are not part of LAPACK).

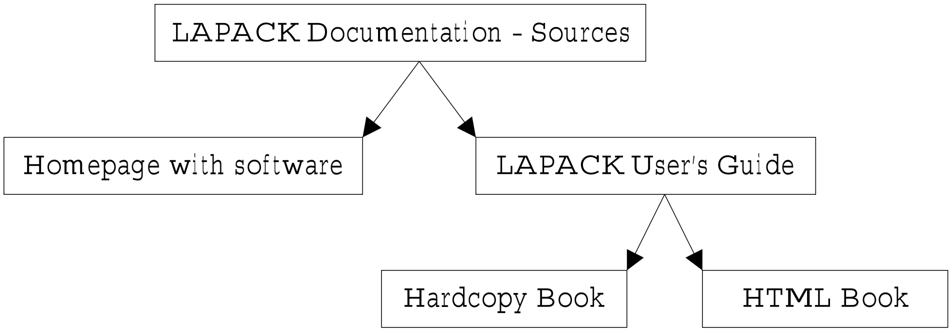
i.e. (approx.)



What are the "important" routines? Well, it depends...

- "...the scientists' goals are to solve the challenging problems..." ⇒ Driver.
- Performance ⇒ Computational. Our work! (?).

Summarizing LAPACK sources and documentation



And "LAWNs" (LAPACK Working Note/s) and A LOT of papers. The web sites:

- LAPACK (1) http://www.netlib.org/lapack
- LAPACK (2) http://www.netlib.org/lapack-dev/lapack-coding/program-style.html
- LAPACK Users' Guide http://www.netlib.org/lapack/lug/index.html
- LAPACK Working Notes (LAWNs) http://www.netlib.org/lapack/lawns/index.html

7 BLAS

From LAPACK Homepage http://www.netlib.org/lapack/

"LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). While LINPACK and EISPACK are based on the vector operations kernels of the Level 1 BLAS, LAPACK was designed at the outset to exploit the L3 BLAS – a set of specifications for FORTRAN subprograms that do various types of matrix multiplication and the solution of triangular systems with multiple right-hand sides"

Finally, almost everything is written in terms of the BLAS and most of the whole performance depends on the L3 BLAS performance.

From BLAS Homepage http://www.netlib.org/blas/

"This material is based upon work supported by the National Science Foundation under Grant No. ASC-9313958 and DOE Grant No. DE-FG03-94ER25219. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF) or the Department of Energy (DOE)."

So??? ok, information is on its FAQ (http://www.netlib.org/blas/faq.html):

1.1) What are the BLAS?
1.2) Publications/references for the BLAS?
1.3) Is there a Quick Reference Guide to the BLAS available?
1.4) Are optimized BLAS libraries available?
1.5) What is ATLAS?
1.6) Where can I find vendor supplied BLAS?
1.7) Where can I find the Intel BLAS for Linux?
1.8) Where can I find Java BLAS?
1.9) Is there a C interface to the BLAS?
1.10) Are prebuilt Fortran77 ref implementation BLAS libraries available from Netlib?

What kind of operations are included? Every operation is included in one "Level", as described in the next section.

7.1 BLAS Levels

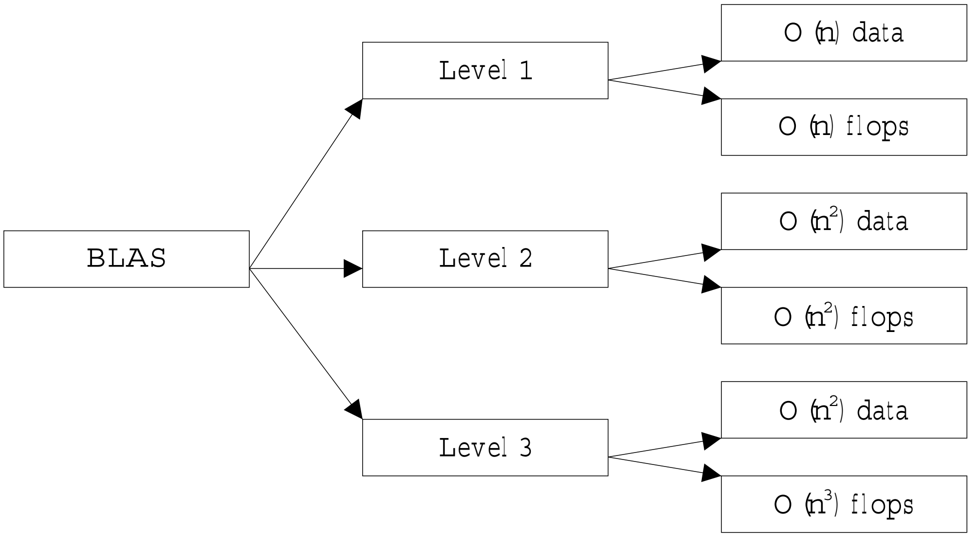
Subroutines included in BLAS are classified according to its requirements of memory and floating point operations. Assuming

- α and β are scalars.
- x and y are n-elements vectors.
- A, B, and C are square matrices with $n \times n$ elements.

The BLAS are divided in three *levels*:

- Level 1 or L1 BLAS: routines making operations among vectors, thus with access to $O(n)$ data and number of floating point operations also $O(n)$, such as $y = \alpha x + y$.
- Level 2 or L2 BLAS: routines making operations among vectors and matrices, thus with access to $O(n^2)$ data and number of floating point operations also $O(n^2)$, such as $y = \alpha Ax + \beta y$.
- Level 3 or L3 BLAS: routines making operations among matrices, thus with access to $O(n^2)$ data and number of floating point operations $O(n^3)$, such as $C = \alpha AB + \beta C$.

Summaryzing:



where flops: "number of floating point operations". Note that

- L1 and L2: 1 flop per access.
- L3: n flops per access?!

And this is why

"Finally, almost everything is written in terms of the BLAS and most of the whole performance depends on the L3 BLAS performance."
(from the previous page).

7.2 Level 3 BLAS

Given that "...most of the whole performance depends on the L3 BLAS performance" it is worth analyzing L3 BLAS in deeper detail. The subroutines defined in this level are:

1) "General" Matrix Multiplication, or matrix multiplication with "general" matrices (_GEMM)

$$C \leftarrow \alpha \; Op(A) \; Op(B) \; + \; \beta \; C$$

where A, B, and C are matrices, α and β are scalars, and $Op(X)$ may be X, X^T or X^H.

Interesting... But how do you make $C = A \times B$?

2) Matrix multiplication involving a symmetric or Hermitian matrix (_SYMM or _HEMM)

$$C \leftarrow \alpha AB + \beta C \qquad or \qquad C \leftarrow \alpha BA + \beta C$$

where matrix A is symmetric (_SYMM) or Hermitian (_HEMM) and is multiplied at left or right of matrix B depending upon a parameter.

3) Matrix multiplication involving a triangular matrix (_TRMM)

$$B \leftarrow \alpha Op(A)B \qquad or \qquad B \leftarrow \alpha BOp(A)$$

where matrix A is triangular, $Op(X)$ may be X, X^T or X^H, and $Op(A)$ is multiplied at left or right of matrix B depending upon a parameter.

4) Rank-k update of a symmetric or Hermitian matrix (_SYRK or _HERK)

$$C \leftarrow \alpha AOp(A) + \beta C \qquad or \qquad C \leftarrow \alpha Op(A)A + \beta C$$

where matrix A is symmetric (_SYRK) or Hermitian (_HERK), if A is symmetric then $Op(A) = A^T$ and $Op(A) = A^H$ otherwise, and A is multiplied at left or right of matrix B depending upon a parameter.

5) Rank-2k update of a symmetric or Hermitian matrix (_SYR2K or _HER2K)

$$C \leftarrow \alpha AOp(B) + \overline{\alpha} BOp(A) + \beta C \qquad or \qquad C \leftarrow \alpha Op(A)B + \overline{\alpha} Op(B)A + \beta C$$

where
if matrix C is symmetric (_SYR2K) then $Op(X) = X^T$ and $\alpha \in \mathbb{R}$, thus $\overline{\alpha} = \alpha$,
if matrix C is Hermitian (_HER2K) then $Op(X) = X^H$,
and A is multiplied at left or right of matrix B depending upon a parameter.

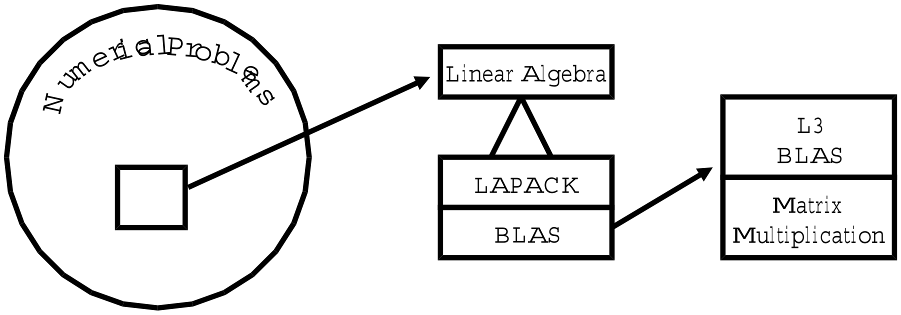
6) Solution of triangular systems of equations with multiple right-hand sides (_TRSM)

$$Op(A)X = \alpha B \qquad or \qquad XOp(A) = \alpha B$$

where matrix A is lower or upper triangular (eventually with unit diagonal), $Op(A)$ may be A, A^T or A^H, and $Op(A)$ is multiplied at left or right of matrix X depending upon a parameter. The matrix X is overwritten on B.

7.3 Wait, Wait, What about Matrix Multiplication?

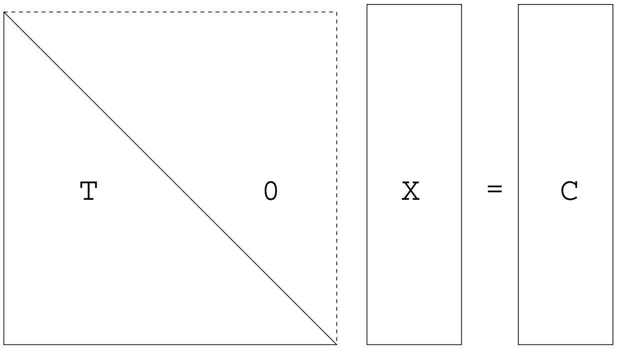
Good question, because

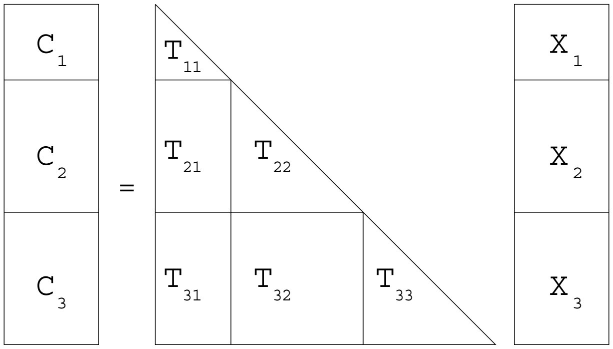


Well, from _GEMM to _XXR2K -items 1) to 5) in the previous enumeration of L3 BLAS description- subroutines are basically matrix multiplications. How are related solutions of triangular systems of equations to matrix multiplications?

Kågström B., P. Ling, C. Van Loan, "Portable High-Performance GEMM-based Level 3 BLAS", R. F. Sincovec et al., Editor, Parallel Processing for Scientific Computing, Philadelphia, 1993, SIAM, pp. 339- 346.

An example starting with one possible _TRSM, where some *blocks/submatrices* can be defined (just like in/for the matrix multiplication)





Thus, solving $TX = C$ could be made/defined by (why?)

$$C_1 = T_{11}X_1 \tag{1}$$

$$C_2 = T_{21}X_1 + T_{22}X_2 \tag{2}$$

$$C_3 = T_{31}X_1 + T_{32}X_2 + T_{33}X_3 \tag{3}$$

In fact, the resolution method using blocks/submatrices is basically the same as that of using directly equations and scalars, i.e. *forward substitution*. First, solve Eq.(1) *as usual* for a triangular system of equations:

$$T_{11}X_1 = C_1$$

Thus obtaining the value of X_1 and use it on Eq.(2), which *becomes* another simple triangular system of equations

$$T_{22}X_2 = C_2 - T_{21}X_1 \tag{4}$$

which can be solved again, thus obtaining the value of X_2, and now using the values of X_1 and X_2, Eq.(3) *becomes*

$$T_{33}X_3 = C_3 - T_{31}X_1 - T_{32}X_2 \tag{5}$$

which is another triangular system of equations. So far, nothing new... except for *blocks*... Note that Eq.(4) and Eq.(5) imply multiplying matrices... or not?

Starting again from Eq.(1), Eq.(2), and Eq.(3), and solving the triangular system of equations for X_1, it is possible to define

$$\begin{aligned} C_2 - T_{21}X_1 &= T_{22}X_2 && (6) \\ C_3 - T_{31}X_1 &= T_{32}X_2 + T_{33}X_3 && (7) \end{aligned}$$

And this system of equations has very important characteristics:

- Solves the original system of equations.
- Can be solved applying the *same* procedure: obtain a the values for X_2 and solve the rest of equations.
- Has two *simultaneous* matrix multiplications.

Where/How are the matrix multiplications?

$$\begin{aligned} C_2 &\leftarrow C_2 - T_{21}X_1 \\ C_3 &\leftarrow C_3 - T_{31}X_1 \end{aligned}$$

Ups! It's just like the _GEMM already defined...

$$C \leftarrow \alpha\ Op(A)\ Op(B)\ +\ \beta\ C$$

Do you see the relationship? ...

7.4 Matrix Multiplication, L3 BLAS, and Performance

Given that "...most of the whole performance depends on the L3 BLAS performance" and everything in L3 BLAS may be GEMM-based, the BIG question is:

Matrix multiplication performance is the whole performance to be obtained by the applications?

By the way: What about parallel performance? And parallel performance on clusters? Are these questions relevant in this context?

Going back to libraries: Examples from LAPACK and BLAS in the next sections. Remember that HPC on linear algebra applications imply using libraries such as LAPACK and/or BLAS.

8 Examples from LAPACK and (L3) BLAS

From LAPACK man pages:

```
SGETRF(1)                      )                        SGETRF(1)



NAME
       SGETRF  -  compute an LU factorization of a general M-by-N
       matrix A using partial pivoting with row interchanges

SYNOPSIS
       SUBROUTINE SGETRF( M, N, A, LDA, IPIV, INFO )

           INTEGER        INFO, LDA, M, N

           INTEGER        IPIV( * )

           REAL           A( LDA, * )

PURPOSE
       SGETRF computes an LU factorization of  a  general  M-by-N
       matrix  A  using  partial  pivoting with row interchanges.
       The factorization has the form
          A = P * L * U
       where P is a permutation matrix,  L  is  lower  triangular
       with  unit diagonal elements (lower trapezoidal if m > n),
       and U is upper triangular (upper trapezoidal if m < n).

       This is the right-looking Level  3  BLAS  version  of  the
       algorithm.


ARGUMENTS
       M       (input) INTEGER
               The number of rows of the matrix A.  M >= 0.

       N       (input) INTEGER
               The number of columns of the matrix A.  N >= 0.

       A       (input/output) REAL array, dimension (LDA,N)
               On  entry,  the  M-by-N matrix to be factored.  On
               exit, the factors L and U from the factorization A
               =  P*L*U;  the unit diagonal elements of L are not
               stored.

       LDA     (input) INTEGER
               The leading dimension of  the  array  A.   LDA  >=
          max(1,M).

 IPIV     (output) INTEGER array, dimension (min(M,N))
          The  pivot  indices; for 1 <= i <= min(M,N), row i
          of the matrix was interchanged with row IPIV(i).

 INFO     (output) INTEGER
          = 0:  successful exit
          < 0:  if INFO = -i, the i-th argument had an ille-
          gal value
          >  0:   if  INFO  = i, U(i,i) is exactly zero. The
          factorization has been completed, but the factor U
          is  exactly  singular,  and  division by zero will
          occur if it is used to solve  a  system  of  equa-
          tions.



LAPACK version 3.0         15 June 2000              SGETRF(1)
```

Two details: `LDA` and `REAL A( LDA, * )`. Well... the third detail: name? From LAPACK online documentation:

Naming Scheme

The name of each LAPACK routine is a coded specification of its function (within the very tight limits of standard Fortran 77 6-character names).

All driver and computational routines have names of the form XYYZZZ, where for some driver routines the 6th character is blank.

The first letter, X, indicates the data type as follows:

S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16 or DOUBLE COMPLEX

When we wish to refer to a LAPACK routine generically, regardless of data type, we replace the first letter by “x”. Thus xGESV refers to any or all of the routines SGESV, CGESV, DGESV and ZGESV.

The next two letters, YY, indicate the type of matrix (or of the most significant matrix). Most of these two-letter codes apply to both real and complex matrices; a few apply specifically to one or the other, as indicated in Table 2.1.

Table 2.1: Matrix types in the LAPACK naming scheme

BD	bidiagonal
DI	diagonal
GB	general band
GE	general (i.e., unsymmetric, in some cases rectangular)
GG	general matrices, generalized problem (i.e., a pair of general matrices)
GT	general tridiagonal
HB	(complex) Hermitian band
HE	(complex) Hermitian

HG upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a triangular matrix)

HP (complex) Hermitian, packed storage

HS upper Hessenberg

OP (real) orthogonal, packed storage

OR (real) orthogonal

PB symmetric or Hermitian positive definite band

PO symmetric or Hermitian positive definite

PP symmetric or Hermitian positive definite, packed storage

PT symmetric or Hermitian positive definite tridiagonal

SB (real) symmetric band

SP symmetric, packed storage

ST (real) symmetric tridiagonal

SY symmetric

TB triangular band

TG triangular matrices, generalized problem (i.e., a pair of triangular matrices)

TP triangular, packed storage

TR triangular (or in some cases quasi-triangular)

TZ trapezoidal

UN (complex) unitary

UP (complex) unitary, packed storage

When we wish to refer to a class of routines that performs the same function on different types of matrices, we replace the first three letters by “xyy”. Thus xyySVX refers to all the expert driver routines for systems of linear equations that are listed in Table 2.2.

The last three letters ZZZ indicate the computation performed. Their meanings will be explained in Section 2.4. For example, SGEBRD is a single precision routine that performs a bidiagonal reduction (BRD) of a real general matrix.

The names of auxiliary routines follow a similar scheme except that the 2nd and 3rd characters YY are usually LA (for example, SLASCL or CLARFG). There are two kinds of exception. Auxiliary routines that implement an unblocked version of a block algorithm have similar names to the routines that perform the block algorithm, with the sixth character being “2” (for example, SGETF2 is the unblocked version of SGETRF). A few routines that may be regarded as extensions to the BLAS are named according to the BLAS naming schemes (for example, CROT, CSYR).

And, finally, on “Section 2.4”

xyyTRF: factorize (obviously not needed for triangular matrices);

From BLAS man pages:

```
SGEMM(1)                   BLAS routine                   SGEMM(1)



NAME
       SGEMM - perform one of the matrix-matrix operations   C :=
       alpha*op( A )*op( B ) + beta*C,

SYNOPSIS
       SUBROUTINE SGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
                          B, LDB, BETA, C, LDC )

           CHARACTER*1  TRANSA, TRANSB

           INTEGER      M, N, K, LDA, LDB, LDC

           REAL         ALPHA, BETA

           REAL         A( LDA, * ), B( LDB, * ), C( LDC, * )

PURPOSE
       SGEMM  performs one of the matrix-matrix operations

       where  op( X ) is one of

          op( X ) = X   or   op( X ) = X',

       alpha  and  beta are scalars, and A, B and C are matrices,
       with op( A ) an m by k matrix,  op( B )  a  k by n  matrix
       and  C an m by n matrix.


PARAMETERS
       TRANSA - CHARACTER*1.  On entry, TRANSA specifies the form
       of op( A ) to be used in the matrix multiplication as fol-
       lows:

       TRANSA = 'N' or 'n',  op( A ) = A.

       TRANSA = 'T' or 't',  op( A ) = A'.

       TRANSA = 'C' or 'c',  op( A ) = A'.

       Unchanged on exit.

       TRANSB - CHARACTER*1.  On entry, TRANSB specifies the form
       of op( B ) to be used in the matrix multiplication as fol-
```

lows:

TRANSB = 'N' or 'n', op(B) = B.

TRANSB = 'T' or 't', op(B) = B'.

TRANSB = 'C' or 'c', op(B) = B'.

Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of the matrix op(A) and of the matrix C. M must be at least zero. Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of the matrix op(B) and the number of columns of the matrix C. N must be at least zero. Unchanged on exit.

K - INTEGER.
On entry, K specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B). K must be at least zero. Unchanged on exit.

ALPHA - REAL.
On entry, ALPHA specifies the scalar alpha. Unchanged on exit.

A - REAL array of DIMENSION (LDA, ka), where ka is k when TRANSA = 'N' or 'n', and is m otherwise. Before entry with TRANSA = 'N' or 'n', the leading m by k part of the array A must contain the matrix A, otherwise the leading k by m part of the array A must contain the matrix A. Unchanged on exit.

LDA - INTEGER.
On entry, LDA specifies the first dimension of A as declared in the calling (sub) program. When TRANSA = 'N' or 'n' then LDA must be at least max(1, m), otherwise LDA must be at least max(1, k). Unchanged on exit.

B - REAL array of DIMENSION (LDB, kb), where kb is n when TRANSB = 'N' or 'n', and is k other-

```
          wise.  Before entry with  TRANSB = 'N' or 'n',  the
          leading  k by n part of the array  B  must contain
          the matrix  B,  otherwise the leading  n by k  part
          of  the  array  B  must  contain  the  matrix B.
          Unchanged on exit.

     LDB    - INTEGER.
          On entry, LDB specifies the first dimension of B as
          declared in the calling (sub) program. When  TRANSB
          = 'N' or 'n' then LDB must be at least  max(  1,  k
          ),  otherwise  LDB  must be at least  max( 1, n ).
          Unchanged on exit.

     BETA   - REAL.
          On entry,  BETA  specifies the scalar  beta.   When
          BETA  is supplied as zero then C need not be set on
          input.  Unchanged on exit.

     C      - REAL             array of DIMENSION ( LDC, n ).
          Before entry, the leading  m  by  n   part  of  the
          array   C  must contain the matrix  C,  except when
          beta  is zero, in which case C need not be  set  on
          entry.   On  exit,  the array  C  is overwritten by
          the  m by n  matrix (  alpha*op(  A  )*op(  B  )  +
          beta*C ).

     LDC    - INTEGER.
          On entry, LDC specifies the first dimension of C as
          declared  in  the  calling   (sub)  program.   LDC
          must   be   at   least  max(  1, m ).  Unchanged on
          exit.

          Level 3 Blas routine.

          --  Written  on  8-February-1989.   Jack  Dongarra,
          Argonne  National Laboratory.  Iain Duff, AERE Har-
          well.  Jeremy Du Croz, Numerical  Algorithms  Group
          Ltd.   Sven  Hammarling, Numerical Algorithms Group
          Ltd.




BLAS routine           16 October 1992              SGEMM(1)
```

Now it's possible to explain why the parameters LDA, LDB, and LDC are necessary... Hint: block processing.

In fact, man pages for LAPACK and BLAS routines show the Fortran *port* of the libraries. Assuming the declarations of

```
  ! square matrices
  REAL, DIMENSION(n, n):: a, b, c
```

How would be the call to SGEMM to make a = b x c?, remember

```
  SUBROUTINE SGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA,
                     B, LDB, BETA, C, LDC )
```

Fortran is fine, but... are there other *ports*? There is a C interface to the BLAS defined in the BLAS Technical Forum Standard http://www.netlib.org/blas/blast-forum/. Most of the material is covered in http://www.netlib.org/blas/blast-forum/cinterface.pdf as well as the *ultimate* C BLAS reference, the file cblas.h, which can be obtained in the same site, at http://www.netlib.org/blas/blast-forum/cblas.h

The Fortran SGEMM *becomes* the C function cblas_sgemm as explained in the file cinterface.pdf, which is included in cblas.h

```
void cblas_sgemm(const enum CBLAS_ORDER Order,
                 const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB,
                 const int M, const int N, const int K,
                 const float alpha, const float *A,
                 const int lda, const float *B, const int ldb,
                 const float beta, float *C, const int ldc);
```

Note that except for the first parameter, parameters are almost the same as those for the Fortran subroutine SGEMM. Furthermore, definitions of enum types are in the file cblas.h too

```
/*
 * Enumerated and derived types
 */
#define CBLAS_INDEX size_t  /* this may vary between platforms */
enum CBLAS_ORDER     {CblasRowMajor=101, CblasColMajor=102};
enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};
...
```

Having all of these definitions... How would be the call to SGEMM to make a = b x c in the C *port* of BLAS or cblas?

Why is BLAS more than a *good idea* for software development and/or subroutines study/definition/classification? Some ideas behind BLAS implementations:

- Almost every hardware vendor has its own BLAS implementation: MKL (Intel Matrix Kernel Library), ACML (AMD Core Math Library). In fact, these are libraries which *include* BLAS.

- There are other libraries, not associated to any hardware vendor, such as ATLAS (Automatically Tuned Linear Algebra Software), which also includes BLAS.

A little and *home made* performance comparison via DGEMM (3000 × 3000 elements) on an AMD Athlon +3000 with Linux 32 bits and ifort

Impl.	**Mflop/s**
No Opt	104
ATLAS	2.429
ACML	2.773
MKL	2.524

DGEMM Comparison for Different Implementations

More on performance from Intel: http://www.intel.com/cd/software/products/asmo-na/eng/266858.htm, look at the threads specification/s.

9 A Step Forward: ScaLAPACK

From ScaLAPACK homepage http://www.netlib.org/scalapack

"The ScaLAPACK project was a collaborative effort involving several institutions:

- Oak Ridge National Laboratory
- Rice University
- University of California, Berkeley
- University of California, Los Angeles
- University of Illinois
- University of Tennessee, Knoxville

and comprised four components:

- dense and band matrix software (ScaLAPACK)
- large sparse eigenvalue software (PARPACK and ARPACK)
- sparse direct systems software (CAPSS and MFACT)
- preconditioners for large sparse iterative solvers (ParPre)

Funding for this effort came in part from DARPA, DOE, NSF, and CRPC."

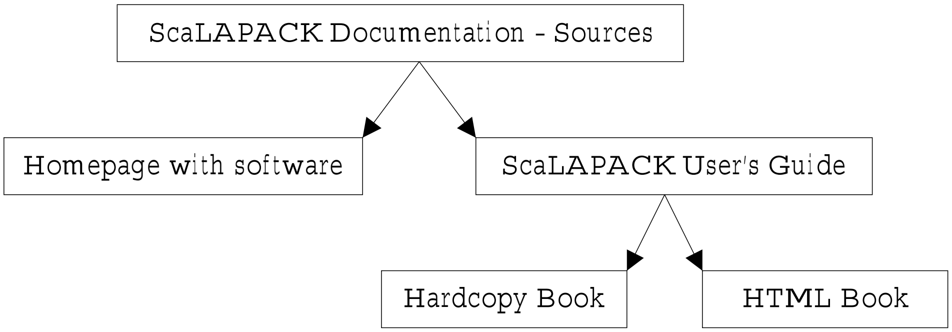
From ScaLAPACK (again) homepage http://www.netlib.org/scalapack/scalapack_home (ok, it is some confusing)

"The ScaLAPACK (or Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition."

This first paragraph has a lot of information:

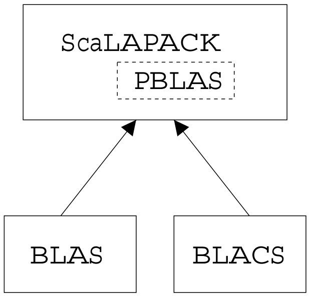
- Subset of LAPACK routines.
- Distributed memory MIMD parallel computers.
- SPMD.
- Explicit message passing.
- Matrices data distribution!

Most of the documentation (and much of the code) is like LAPACK. In fact, the "graphical view" of the ScaLAPACK sources and documentation is



And some “lawns” (LAPACK Working Note/s) and A LOT of papers.

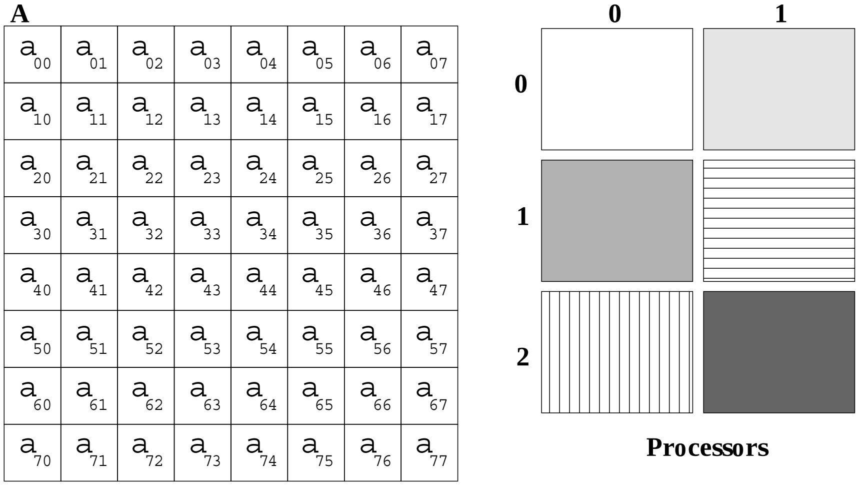
The *official* relationship among ScaLAPACK, LAPACK and BLAS is shown in the ScaLAPACK homepage, which can be summarized as



Added by ScaLAPACK

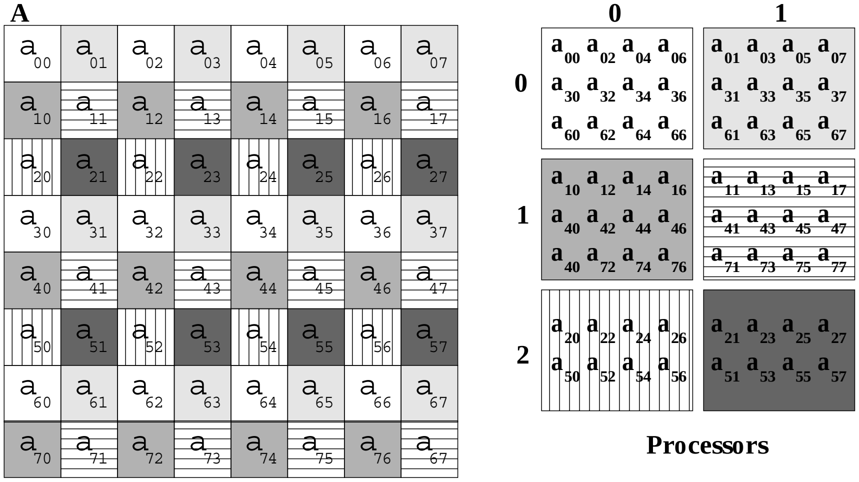
- PBLAS.
- BLACS.

A little introduction to "...two-dimensional block cyclic decomposition..." Having a matrix and a 2-Dimensional array of processors:



A							
a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}

The resulting matrix distribution is



A							
a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{67}

	0	1
0	a_{00} a_{02} a_{04} a_{06} a_{30} a_{32} a_{34} a_{36} a_{60} a_{62} a_{64} a_{66}	a_{01} a_{03} a_{05} a_{07} a_{31} a_{33} a_{35} a_{37} a_{61} a_{63} a_{65} a_{67}
1	a_{10} a_{12} a_{14} a_{16} a_{40} a_{42} a_{44} a_{46} a_{40} a_{72} a_{74} a_{76}	a_{11} a_{13} a_{15} a_{17} a_{41} a_{43} a_{45} a_{47} a_{71} a_{73} a_{75} a_{77}
2	a_{20} a_{22} a_{24} a_{26} a_{50} a_{52} a_{54} a_{56}	a_{21} a_{23} a_{25} a_{27} a_{51} a_{53} a_{55} a_{57}

Reason: matrix factorization algorithms. Why does matrix distribution becomes visible to the user?

What about an example?

```
PSGETRF(l)                          )                                PSGETRF(l)



NAME
       PSGETRF  - compute an LU factorization of a general M-by-N
       distributed matrix sub( A ) = (IA:IA+M-1,JA:JA+N-1)  using
       partial pivoting with row interchanges

SYNOPSIS
       SUBROUTINE PSGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )

           INTEGER         IA, INFO, JA, M, N
```

```
    INTEGER         DESCA( * ), IPIV( * )

    REAL            A( * )
```

PURPOSE

PSGETRF computes an LU factorization of a general M-by-N distributed matrix sub(A) = (IA:IA+M-1,JA:JA+N-1) using partial pivoting with row interchanges. The factorization has the form sub(A) = P * L * U, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m > n), and U is upper triangular (upper trapezoidal if m < n). L and U are stored in sub(A).

This is the right-looking Parallel Level 3 BLAS version of the algorithm.

Notes
=====

Each global data object is described by an associated description vector. This vector stores the information required to establish the mapping between an object element and its corresponding process and memory location.

Let A be a generic term for any 2D block cyclicly distributed array. Such a global array has an associated description vector DESCA. In the following comments, the character _ should be read as "of the global array".

```
NOTATION        STORED IN      EXPLANATION
---------------                              --------------
-------------------------------------     DTYPE_A(global)
DESCA( DTYPE_ )The descriptor type.  In this case,
                               DTYPE_A = 1.
CTXT_A  (global)  DESCA( CTXT_ ) The BLACS context handle,
indicating
                               the BLACS process grid A is
distribu-
                               ted   over.   The   context
itself is glo-
                               bal, but  the  handle  (the
integer
                               value) may vary.
M_A     (global)  DESCA( M_ )    The number of rows in the
global
                               array A.
```

```
       N_A    (global) DESCA( N_ )    The number  of  columns  in
       the global
                                      array A.
       MB_A   (global) DESCA( MB_ )   The blocking factor used to
       distribute
                                      the rows of the array.
       NB_A   (global) DESCA( NB_ )   The blocking factor used to
       distribute
                                      the columns of the array.
       RSRC_A  (global) DESCA( RSRC_ ) The process row over which
       the first
                                      row of the array A is  dis-
       tributed.  CSRC_A (global) DESCA( CSRC_ ) The process col-
       umn over which the
                                      first column of the array A
       is
                                      distributed.
       LLD_A   (local)   DESCA(  LLD_ )  The leading dimension of
       the local
                                      array.       LLD_A       >=
       MAX(1,LOCr(M_A)).

       Let  K  be  the number of rows or columns of a distributed
       matrix, and assume that its process grid has dimension p x
       q.
       LOCr(  K ) denotes the number of elements of K that a pro-
       cess would receive if K were distributed over the  p  pro-
       cesses of its process column.
       Similarly,  LOCc(  K ) denotes the number of elements of K
       that a process would receive if K  were  distributed  over
       the q processes of its process row.
       The  values  of  LOCr() and LOCc() may be determined via a
       call to the ScaLAPACK tool function, NUMROC:
               LOCr( M ) = NUMROC( M, MB_A, MYROW, RSRC_A,  NPROW
       ),
               LOCc(  N ) = NUMROC( N, NB_A, MYCOL, CSRC_A, NPCOL
       ).  An upper bound for these quantities  may  be  computed
       by:
               LOCr( M ) <= ceil( ceil(M/MB_A)/NPROW )*MB_A
               LOCc( N ) <= ceil( ceil(N/NB_A)/NPCOL )*NB_A

       This  routine requires square block decomposition ( MB_A =
       NB_A ).


ARGUMENTS
       M       (global input) INTEGER
               The number of rows to be  operated  on,  i.e.  the
```

number of rows of the distributed submatrix sub(A). M >= 0.

N (global input) INTEGER
The number of columns to be operated on, i.e. the number of columns of the distributed submatrix sub(A). N >= 0.

A (local input/local output) REAL pointer into the local memory to an array of dimension (LLD_A, LOCc(JA+N-1)). On entry, this array contains the local pieces of the M-by-N distributed matrix sub(A) to be factored. On exit, this array contains the local pieces of the factors L and U from the factorization sub(A) = P*L*U; the unit diagonal ele- ments of L are not stored.

IA (global input) INTEGER
The row index in the global array A indicating the first row of sub(A).

JA (global input) INTEGER
The column index in the global array A indicating the first column of sub(A).

DESCA (global and local input) INTEGER array of dimension DLEN_.
The array descriptor for the distributed matrix A.

IPIV (local output) INTEGER array, dimension (LOCr(M_A)+MB_A)
This array contains the pivoting information. IPIV(i) -> The global row local row i was swapped with. This array is tied to the distributed matrix A.

INFO (global output) INTEGER
= 0: successful exit
< 0: If the i-th argument is an array and the j-entry had an illegal value, then INFO = -(i*100+j), if the i-th argument is a scalar and had an illegal value, then INFO = -i. > 0: If INFO = K, U(IA+K-1,JA+K-1) is exactly zero. The factorization has been completed, but the factor U is exactly singular, and division by zero will occur if it is used to solve a system of equations.

10 Specific Algorithms for Simple Operations

First: there are not complex operations. However, the simplest (and most important?): matrix multiplication. Given

$$A \in I\!R^{m\times k}$$

and

$$B \in I\!R^{k\times n}$$

where the elements of matrix A are denoted as

$$a_{ij}, 1 \le i \le m, 1 \le j \le k$$

and the elements of matrix B are denoted as

$$b_{ij}, 1 \le i \le k, 1 \le j \le n$$

matrix

$$C \in I\!R^{m\times n}$$

with elements denoted as

$$c_{ij}, 1 \le i \le m, 1 \le j \le n$$

from the multiplication

$$C = A \times B$$

is defined by

$$c_{ij} = \sum_{r=1}^{k} a_{ik} b_{kj}$$

If $m = n = k$, the number of floating point operations is $O(n^3)$. Furthermore, the exact number of floating point operations, $flopsMM$, is

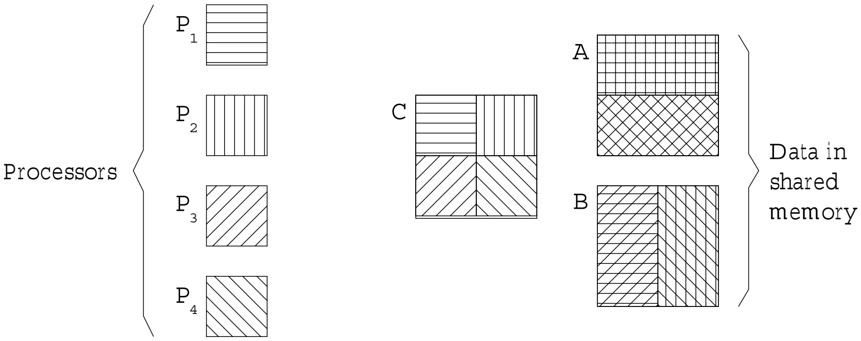
$$flopsMM = 2n^3 - n^2$$

having square matrices of order n. The general parallel matrix multiplication algorithms:

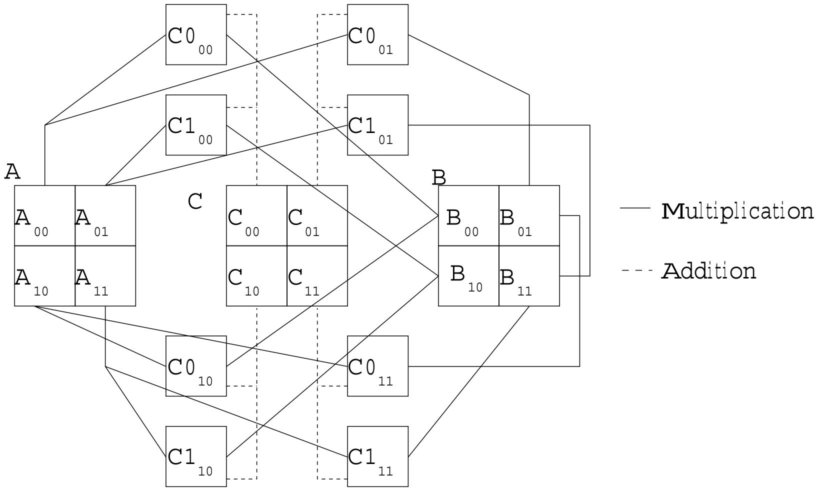
- For multiprocessors (shared memory parallel computers).
- For multicomputers (distributed memory parallel computers).
- For Clusters!

10.1 Matrix Multiplication on Multiprocessors

The simplest



With recursion



```
mat_mul(A, B, C, s) /* C = A×B */
/* A, B: operands */
/* C   : result */
/* s   : matrices (square) size */
{
   if (sequential multiplication)
   {
      C = A×B;
   }
   else
   {
      mat_mul(A00, B00, C000, s/2);   /* (1) */
      mat_mul(A01, B10, C100, s/2);   /* (2) */
      mat_mul(A00, B01, C001, s/2);   /* (3) */
      mat_mul(A01, B11, C101, s/2);   /* (4) */
      mat_mul(A10, B00, C010, s/2);   /* (5) */
      mat_mul(A11, B10, C110, s/2);   /* (6) */
      mat_mul(A10, B01, C011, s/2);   /* (7) */
      mat_mul(A11, B11, C111, s/2);   /* (8) */
   }
   C00 = C000 + C100;
   C01 = C001 + C101;
   C10 = C010 + C110;
   C11 = C011 + C111;
}
```

Is $C_{00} = C0_{00} + C1_{00}$?

$$\begin{array}{cc|cc}
\multicolumn{2}{c}{C_{00}} & \multicolumn{2}{c}{C_{01}} \\ \hline
c_{00} & c_{01} & c_{02} & c_{03} \\
c_{10} & c_{11} & c_{12} & c_{13} \\ \hline
c_{20} & c_{21} & c_{22} & c_{23} \\
c_{30} & c_{31} & c_{32} & c_{33} \\ \hline
\multicolumn{2}{c}{C_{10}} & \multicolumn{2}{c}{C_{11}}
\end{array}
=
\begin{array}{cc|cc}
\multicolumn{2}{c}{A_{00}} & \multicolumn{2}{c}{A_{01}} \\ \hline
a_{00} & a_{01} & a_{02} & a_{03} \\
a_{10} & a_{11} & a_{12} & a_{13} \\ \hline
a_{20} & a_{21} & a_{22} & a_{23} \\
a_{30} & a_{31} & a_{32} & a_{33} \\ \hline
\multicolumn{2}{c}{A_{10}} & \multicolumn{2}{c}{A_{11}}
\end{array}
\text{ x }
\begin{array}{cc|cc}
\multicolumn{2}{c}{B_{00}} & \multicolumn{2}{c}{B_{01}} \\ \hline
b_{00} & b_{01} & b_{02} & b_{03} \\
b_{10} & b_{11} & b_{12} & b_{13} \\ \hline
b_{20} & b_{21} & b_{22} & b_{23} \\
b_{30} & b_{31} & b_{32} & b_{33} \\ \hline
\multicolumn{2}{c}{B_{10}} & \multicolumn{2}{c}{B_{11}}
\end{array}$$

$$\mathrm{C0}_{00} = \mathrm{A}_{00} \text{ x } \mathrm{B}_{00}$$
$$\mathrm{C1}_{00} = \mathrm{A}_{01} \text{ x } \mathrm{B}_{10}$$

$$\mathrm{C0}_{00} = \boxed{\begin{array}{cc} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{array}}$$

$$\mathrm{C1}_{00} = \boxed{\begin{array}{cc} a_{02}b_{20} + a_{03}b_{30} & a_{02}b_{21} + a_{03}b_{31} \\ a_{12}b_{20} + a_{13}b_{30} & a_{12}b_{21} + a_{13}b_{31} \end{array}}$$

Thus, $\mathrm{C0}_{00} + \mathrm{C1}_{00}$

$$\boxed{\begin{array}{cc} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} + a_{13}b_{30} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \end{array}}$$

Important: this is block processing.

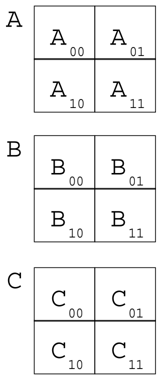
Avoiding extra memory requirements and including data dependence

```
mat_mul_sum(A, B, C, s) /* C = A×B + C */
/* A, B: operands */
/* C   : result */
/* s   : matrices (square) size */
{
   if (sequential multiplication)
   {
      C = A×B + C;
   }
   else
   {
      mat_mul_sum(A00, B00, C00, s/2);    /* (1) */
      mat_mul_sum(A01, B10, C00, s/2);    /* (2) */
      mat_mul_sum(A00, B01, C01, s/2);    /* (3) */
      mat_mul_sum(A01, B11, C01, s/2);    /* (4) */
      mat_mul_sum(A10, B00, C10, s/2);    /* (5) */
      mat_mul_sum(A11, B10, C10, s/2);    /* (6) */
      mat_mul_sum(A10, B01, C11, s/2);    /* (7) */
      mat_mul_sum(A11, B11, C11, s/2);    /* (8) */
   }
}
```

Strassen's Method (works? flop count?)

$$P_0 = (A_{00} + A_{11}) \times (B_{00} + B_{11})$$
$$P_1 = (A_{10} + A_{11}) \times B_{00}$$
$$P_2 = A_{00} \times (B_{01} - B_{11})$$
$$P_3 = A_{11} \times (B_{10} - B_{00})$$
$$P_4 = (A_{00} + A_{01}) \times B_{11}$$
$$P_5 = (A_{10} - A_{00}) \times (B_{00} + B_{01})$$
$$P_6 = (A_{01} - A_{11}) \times (B_{10} + B_{11})$$
$$C_{00} = P_0 + P_3 - P_4 + P_6$$
$$C_{01} = P_2 + P_4$$
$$C_{10} = P_1 + P_3$$
$$C_{11} = P_0 + P_3 - P_1 + P_5$$

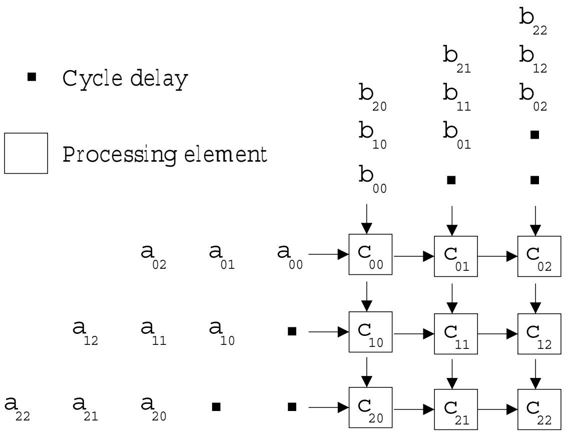
a) Computing with submatrices

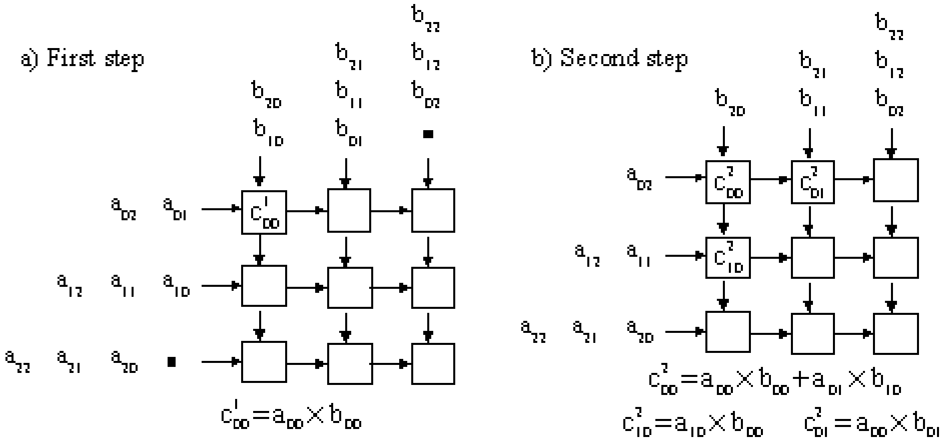


b) Matrix Partition

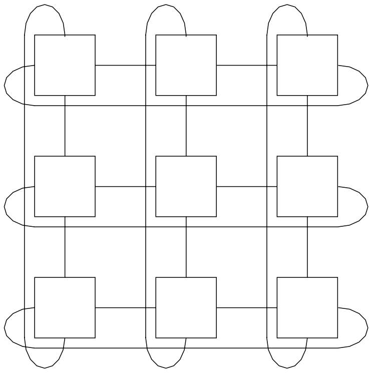
10.2 Matrix Multiplication on Multicomputers

On distributed memory parallel computers: basically Cannon & Fox. However,

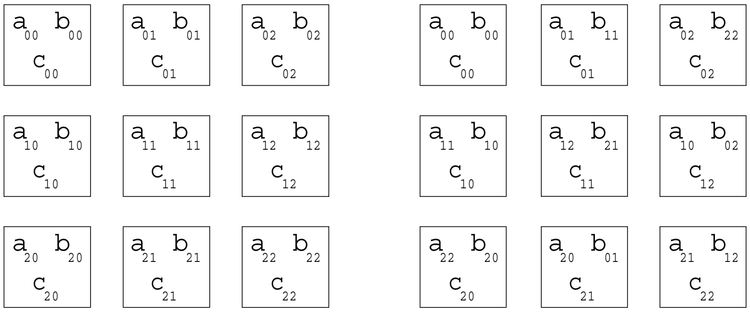



a) First step
b) Second step
c^1_{00}
$c^1_{00}=a_{00}\times b_{00}$
c^2_{00} c^2_{01} c^2_{10}
$c^2_{00}=a_{00}\times b_{00}+a_{01}\times b_{10}$
$c^2_{10}=a_{10}\times b_{00}$
$c^2_{01}=a_{00}\times b_{01}$

The most common interprocessor network

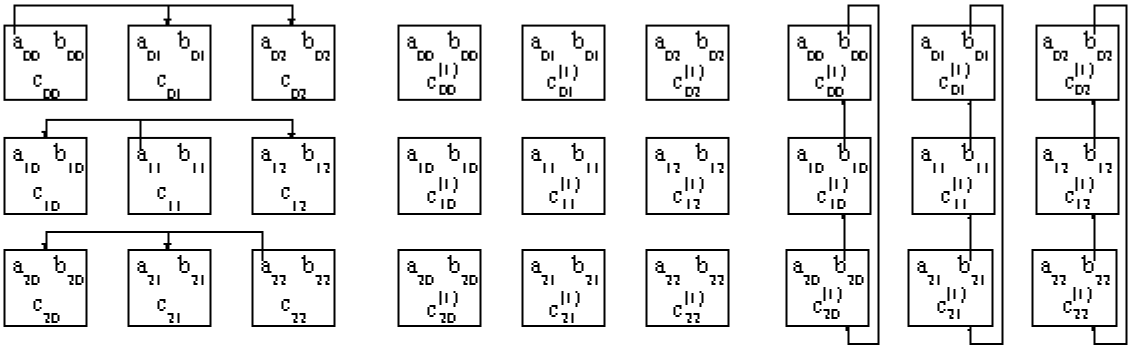


Cannon: relocation and shifts



a) Initial Distribution a) Initial Alignment

Fox: broadcast and shift

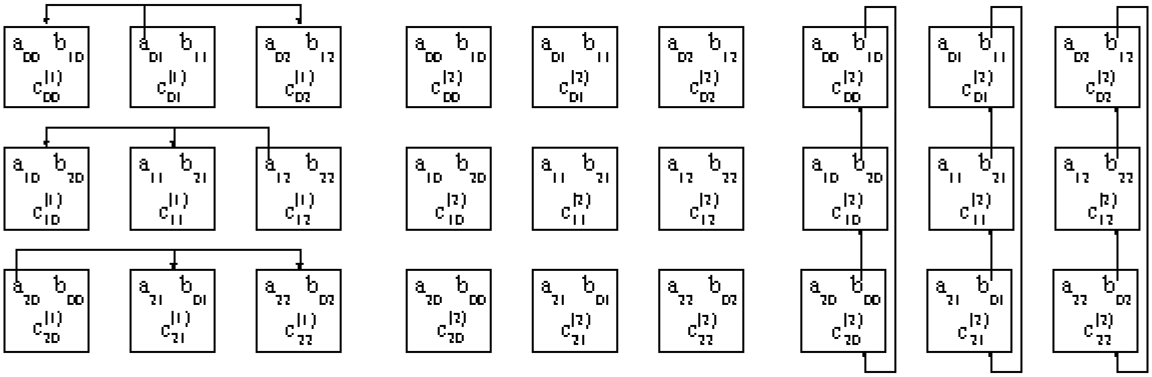


$$c_{00}^{(1)}=a_{00}\times b_{00};\ c_{01}^{(1)}=a_{00}\times b_{01};\ c_{02}^{(1)}=a_{00}\times b_{02}$$

$$c_{10}^{(1)}=a_{11}\times b_{10};\ c_{11}^{(1)}=a_{11}\times b_{11};\ c_{12}^{(1)}=a_{11}\times b_{12}$$

$$c_{20}^{(1)}=a_{22}\times b_{20};\ c_{21}^{(1)}=a_{22}\times b_{21};\ c_{22}^{(1)}=a_{22}\times b_{22}$$

a) Broadcast b) Local computing a) Shift



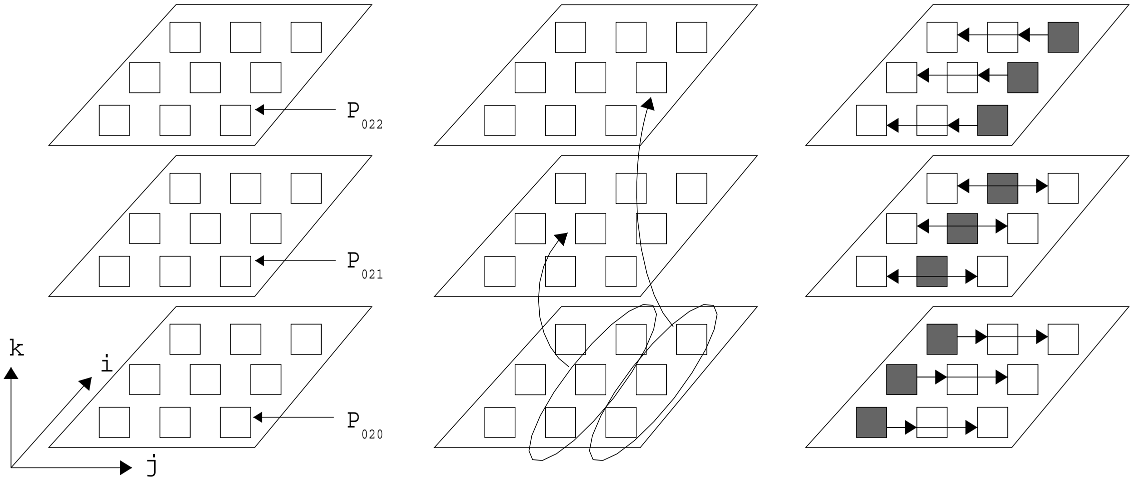
$$c_{00}^{(2)}=c_{00}^{(1)}+a_{01}\times b_{10};\ c_{01}^{(2)}=c_{01}^{(1)}+a_{01}\times b_{11};\ c_{02}^{(2)}=c_{02}^{(1)}+a_{01}\times b_{12}$$

$$c_{10}^{(2)}=c_{10}^{(1)}+a_{12}\times b_{20};\ c_{11}^{(2)}=c_{11}^{(1)}+a_{12}\times b_{21};\ c_{12}^{(2)}=c_{12}^{(1)}+a_{12}\times b_{22}$$

$$c_{20}^{(2)}=c_{20}^{(1)}+a_{20}\times b_{00};\ c_{21}^{(2)}=c_{21}^{(1)}+a_{20}\times b_{01};\ c_{22}^{(2)}=c_{22}^{(1)}+a_{20}\times b_{02}$$

a) Broadcast b) Local computing a) Shift

Three dimensions and more: DNS (data replication)



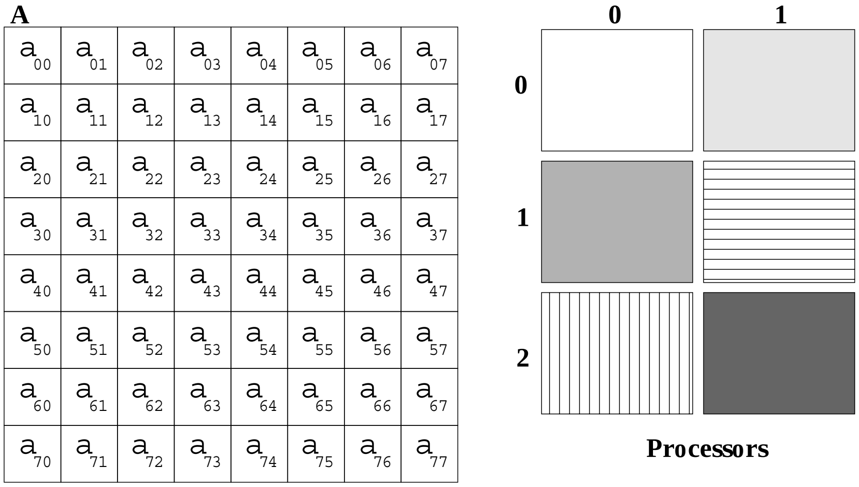
a) Processors identification. b) Distribution of columns of A. c) Broadcast by Planes.

10.3 Matrix Multiplication on Clusters

Some previous work first, which is based on previous algorithms for multicomputers. Later, the specific proposal for (Ethernet-interconnected) clusters.

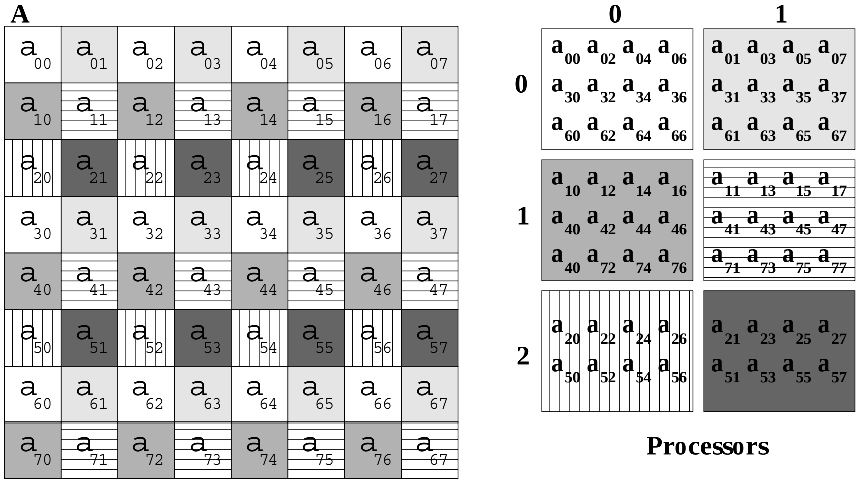
10.4 Matrix Multiplication Already Proposed for DMPC (and Clusters)

PUMMA-SUMMA-DIMMA (MMA: Matrix Multiplication Algorithm). SUMMA: Scalable Universal MMA, almost directly used in ScaLAPACK. Taking into account "two-dimensional block cyclic decomposition" is losely based on Fox's algorithm and losely resembling Cannon's algorithm data communication pattern. Data distribution:



a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}

The resulting matrix distribution is



A

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}
a_{60}	a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}
a_{70}	a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{67}

	0	**1**
0	$a_{00}\ a_{02}\ a_{04}\ a_{06}$ $a_{30}\ a_{32}\ a_{34}\ a_{36}$ $a_{60}\ a_{62}\ a_{64}\ a_{66}$	$a_{01}\ a_{03}\ a_{05}\ a_{07}$ $a_{31}\ a_{33}\ a_{35}\ a_{37}$ $a_{61}\ a_{63}\ a_{65}\ a_{67}$
1	$a_{10}\ a_{12}\ a_{14}\ a_{16}$ $a_{40}\ a_{42}\ a_{44}\ a_{46}$ $a_{40}\ a_{72}\ a_{74}\ a_{76}$	$a_{11}\ a_{13}\ a_{15}\ a_{17}$ $a_{41}\ a_{43}\ a_{45}\ a_{47}$ $a_{71}\ a_{73}\ a_{75}\ a_{77}$
2	$a_{20}\ a_{22}\ a_{24}\ a_{26}$ $a_{50}\ a_{52}\ a_{54}\ a_{56}$	$a_{21}\ a_{23}\ a_{25}\ a_{27}$ $a_{51}\ a_{53}\ a_{55}\ a_{57}$

Processors

SUMMA in pseudocode (assuming k is the "common" index, columns of A and rows of B) without blocking:

```
for(i = 0; i < k ; i++)
{
  Send k-th column of A in a row broadcast
  Send k-th row of B in a column broadcast
  Multiply k-th column by k-th row
}
```

which uses broadcasts as Fox's algorithm.

The iterations on processor 0:

$$\overrightarrow{Broadcasts}_0 \quad \begin{array}{c|cccc|} & b_{00} & b_{02} & b_{04} & b_{06} \\ \hline a_{00} & c^{(0)}_{00} & c^{(0)}_{02} & c^{(0)}_{04} & c^{(0)}_{06} \\ a_{30} & c^{(0)}_{30} & c^{(0)}_{32} & c^{(0)}_{34} & c^{(0)}_{36} \\ a_{60} & c^{(0)}_{60} & c^{(0)}_{62} & c^{(0)}_{64} & c^{(0)}_{66} \\ \hline \end{array} \qquad \overrightarrow{Broadcasts}_1 \quad \begin{array}{c|cccc|} & b_{10} & b_{12} & b_{14} & b_{16} \\ \hline a_{01} & c^{(1)}_{00} & c^{(1)}_{02} & c^{(1)}_{04} & c^{(1)}_{06} \\ a_{31} & c^{(1)}_{30} & c^{(1)}_{32} & c^{(1)}_{34} & c^{(1)}_{36} \\ a_{61} & c^{(1)}_{60} & c^{(1)}_{62} & c^{(1)}_{64} & c^{(1)}_{66} \\ \hline \end{array}$$

$$\overrightarrow{Broadcasts}_2 \quad \begin{array}{c|cccc|} & b_{20} & b_{22} & b_{24} & b_{26} \\ \hline a_{02} & c^{(2)}_{00} & c^{(2)}_{02} & c^{(2)}_{04} & c^{(2)}_{06} \\ a_{32} & c^{(2)}_{30} & c^{(2)}_{32} & c^{(2)}_{34} & c^{(2)}_{36} \\ a_{62} & c^{(2)}_{60} & c^{(2)}_{62} & c^{(2)}_{64} & c^{(2)}_{66} \\ \hline \end{array} \qquad \overrightarrow{Broadcasts}_3 \quad \begin{array}{c|cccc|} & b_{30} & b_{32} & b_{34} & b_{36} \\ \hline a_{03} & c^{(3)}_{00} & c^{(3)}_{02} & c^{(3)}_{04} & c^{(3)}_{06} \\ a_{33} & c^{(3)}_{30} & c^{(3)}_{32} & c^{(3)}_{34} & c^{(3)}_{36} \\ a_{63} & c^{(3)}_{60} & c^{(3)}_{62} & c^{(3)}_{64} & c^{(3)}_{66} \\ \hline \end{array}$$

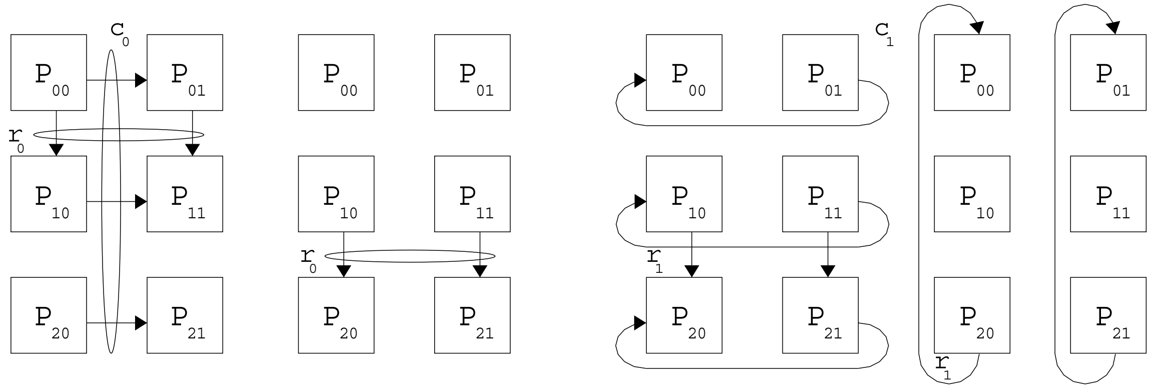
$$\overrightarrow{Broadcasts}_4 \quad \begin{array}{c|cccc|} & b_{40} & b_{42} & b_{44} & b_{46} \\ \hline a_{04} & c^{(4)}_{00} & c^{(4)}_{02} & c^{(4)}_{04} & c^{(4)}_{06} \\ a_{34} & c^{(4)}_{30} & c^{(4)}_{32} & c^{(4)}_{34} & c^{(4)}_{36} \\ a_{64} & c^{(4)}_{60} & c^{(4)}_{62} & c^{(4)}_{64} & c^{(4)}_{66} \\ \hline \end{array} \qquad \overrightarrow{Broadcasts}_5 \quad \begin{array}{c|cccc|} & b_{50} & b_{52} & b_{54} & b_{56} \\ \hline a_{05} & c^{(5)}_{00} & c^{(5)}_{02} & c^{(5)}_{04} & c^{(5)}_{06} \\ a_{35} & c^{(5)}_{30} & c^{(5)}_{32} & c^{(5)}_{34} & c^{(5)}_{36} \\ a_{65} & c^{(5)}_{60} & c^{(5)}_{62} & c^{(5)}_{64} & c^{(5)}_{66} \\ \hline \end{array}$$

$$\overrightarrow{Broadcasts}_6 \quad \begin{array}{c|cccc|} & b_{60} & b_{62} & b_{64} & b_{66} \\ \hline a_{06} & c^{(6)}_{00} & c^{(6)}_{02} & c^{(6)}_{04} & c^{(6)}_{06} \\ a_{36} & c^{(6)}_{30} & c^{(6)}_{32} & c^{(6)}_{34} & c^{(6)}_{36} \\ a_{66} & c^{(6)}_{60} & c^{(6)}_{62} & c^{(6)}_{64} & c^{(6)}_{66} \\ \hline \end{array} \qquad \overrightarrow{Broadcasts}_7 \quad \begin{array}{c|cccc|} & b_{70} & b_{72} & b_{74} & b_{76} \\ \hline a_{07} & c^{(7)}_{00} & c^{(7)}_{02} & c^{(7)}_{04} & c^{(7)}_{06} \\ a_{37} & c^{(7)}_{30} & c^{(7)}_{32} & c^{(7)}_{34} & c^{(7)}_{36} \\ a_{67} & c^{(7)}_{60} & c^{(7)}_{62} & c^{(7)}_{64} & c^{(7)}_{66} \\ \hline \end{array}$$

And this is the final result.

(1) Focusing on granularity and local processing performance, matrix blocks or submatrices are used instead of single elements: $x_{ij} \to X_{ij}$. It is not very clear how to define block size (combination of local computing performance and granularity).

(2) Focusing on static interconnection networks (classical on traditional multicomputers), broadcasts are not easily implemented (poor performance is expected *a priori*). Broadcasts are *transformed into* (and the whole algorithm) multiple and pipelined point-to-point messages through the ring of columns or rows.



which resembles Cannon's algorithm.

(3) Focusing on delays produced by the effect of pipelining (note the time at which the first column of processors could send the first message of it's second and third column block), the concept of LCM(P, Q) (Least Common Multiple) is used in DIMMA (Distribution-Independent MMA) and P/Q ratio. DIMMA also defines explicit algorithmic rearrangements depending on k_{opt} and k_b (which is the first one in *recognizing* there is a difference).

An this algorithm is used in ScaLAPACK (PBLAS, more specifically).