Chapter 6

Dense linear algebra: BLAS, LAPACK, SCALAPACK

In this section we will discuss libraries for dense linear algebra operations.

Dense linear algebra, that is linear algebra on matrices that are stored as two-dimensional arrays (as opposed to sparse linear algebra; see *HPC book, section 5.4*, as well as the tutorial on PETSc *Parallel Programming book, part III*) has been standardized for a considerable time. The basic operations are defined by the three levels of *Basic Linear Algebra Subprograms (BLAS)*:

- Level 1 defines vector operations that are characterized by a single loop [14].
- Level 2 defines matrix vector operations, both explicit such as the matrix-vector product, and implicit such as the solution of triangular systems [7].
- Level 3 defines matrix-matrix operations, most notably the matrix-matrix product [6].

The name 'BLAS' suggests a certain amount of generality, but the original authors were clear [14] that these subprograms only covered dense linear algebra. Attempts to standardize sparse operations have never met with equal success.

Based on these building blocks, libraries have been built that tackle the more sophisticated problems such as solving linear systems, or computing eigenvalues or singular values. *Linpack*[1] and *Eispack* were the first to formalize these operations involved, using Blas Level 1 and Blas Level 2 respectively. A later development, *Lapack* uses the blocked operations of Blas Level 3. As you saw in section *HPC book, section 1.6.1*, this is needed to get high performance on cache-based CPUs.

Remark *The reference implementation* `https://netlib.org/blas/index.html` *of the BLAS [3] will not give good performance with any compiler; most platforms have vendor-optimized implementations, such as the* MKL *library from Intel.*

With the advent of parallel computers, several projects arose that extended the Lapack functionality to distributed computing, most notably *Scalapack* [4, 2], *PLapack* [25, 24], and most recently Elemental [21]. These packages are harder to use than Lapack because of the need for a two-dimensional cyclic distribution; sections *HPC book, section 7.2.3* and *HPC book, section 7.3.2*. We will not go into the details here.

1. The linear system solver from this package later became the *Linpack benchmark*; see section *HPC book, section 2.11.4*.

6.1 Some general remarks

6.1.1 The Fortran heritage

The original BLAS routines were written in Fortran, and the reference implementation is still in Fortran. For this reason you will see the routine definitions first in Fortran in this tutorial It is possible to use the Fortran routines from a C/C++ program:

- You typically need to append an underscore to the Fortran name;
- You need to include a prototype file in your source, for instance `mkl.h`;
- Every argument needs to be a 'star'-argument, so you can not pass literal constants: you need to pass the address of a variable.
- You need to create a column-major matrix.

There are also C/C++ interfaces:

- The C routine names are formed by prefixing the original name with `cblas_`; for instance `dasum` becomes `cblas_dasum`.
- Fortran character arguments have been replaced by enumerated constants, for instance `CblasNoTrans` instead of the `'N'` parameter.
- The Cblas interface can accommodate both row-major and column-major storage.
- Array indices are `1`-based, rather than `0`-based; this mostly becomes apparent in error messages and when specifying pivot indices.

6.1.2 Routine naming

Routines conform to a general naming scheme: `XYYZZZ` where

X precision: `S,D,C,Z` stand for single and double, single complex and double complex, respectively.
YY storage scheme: general rectangular, triangular, banded.
ZZZ operation. See the manual for a list.

6.1.3 Data formats

Lapack and Blas use a number of data formats, including

GE General matrix: stored two-dimensionally as `A(LDA,*)`
SY/HE Symmetric/Hermitian: general storage; `UPLO` parameter to indicate upper or lower (e.g. `SPOTRF`)
GB/SB/HB General/symmetric/Hermitian band; these formats use column-major storage; in `SGBTRF` overallocation needed because of pivoting
PB Symmetric of Hermitian positive definite band; no overallocation in `SPDTRF`

6.1.4 Lapack operations

For Lapack, we can further divide the routines into an organization with three levels:

- Drivers. These are powerful top level routine for problems such as solving linear systems or computing an SVD. There are simple and expert drivers; the expert ones have more numerical sophistication.

- Computational routines. These are the routines that drivers are built up out of. A user may have occasion to call them by themselves.
- Auxiliary routines.

Expert driver names end on 'X'.

- Linear system solving. Simple drivers: `-SV` (e.g., `DGESV`) Solve $AX = B$, overwrite A with LU (with pivoting), overwrite B with X.
  Expert driver: `-SVX` Also transpose solve, condition estimation, refinement, equilibration
- Least squares problems. Drivers:
  `xGELS` using QR or LQ under full-rank assumption
  `xGELSY` "complete orthogonal factorization"
  `xGELSS` using SVD
  `xGELSD` using divide-conquer SVD (faster, but more workspace than `xGELSS`)
  Also: LSE & GLM linear equality constraint & general linear model
- Eigenvalue routines. Symmetric/Hermitian: `xSY` or `xHE` (also `SP`, `SB`, `ST`)
  simple driver `-EV`
  expert driver `-EVX`
  divide and conquer `-EVD`
  relative robust representation `-EVR`
  General (only `xGE`)
  Schur decomposition `-ES` and `-ESX`
  eigenvalues `-EV` and `-EVX`
  SVD (only `xGE`)
  simple driver `-SVD`
  divide and conquer `SDD`
  Generalized symmetric (`SY` and `HE`; `SP`, `SB`)
  simple driver `GV`
  expert `GVX`
  divide-conquer `GVD`
  Nonsymmetric:
  Schur: simple `GGES`, expert `GGESX`
  eigen: simple `GGEV`, expert `GGEVX`
  svd: `GGSVD`

6.2 BLAS matrix storage

There are a few points to bear in mind about the way matrices are stored in the BLAS and LAPACK[2]:

6.2.1 Array indexing

Since these libraries originated in a Fortran environment, they use 1-based indexing. Users of languages such as C/C++ are only affected by this when routines use index arrays, such as the location of pivots in LU factorizations.

2. We are not going into band storage here.

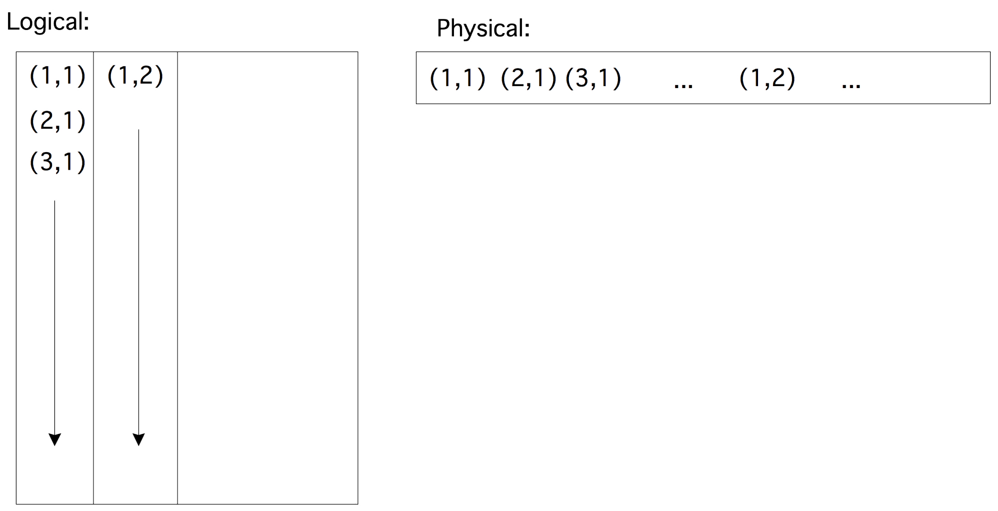

Figure 6.1: Column-major storage of an array in Fortran.

6.2.2 Fortran column-major ordering

Since computer memory is one-dimensional, some conversion is needed from two-dimensional matrix coordinates to memory locations. The *Fortran* language uses *column-major* storage, that is, elements in a column are stored consecutively; see figure 6.1. This is also described informally as 'the leftmost index varies quickest'.

Arrays in C, on the other hand, are laid out in *row-major* order.

6.2.3 Submatrices and the `LDA` parameter

Using the storage scheme described above, it is clear how to store an $m \times n$ matrix in mn memory locations. However, there are many cases where software needs access to a matrix that is a subblock of another, larger, matrix. As you see in figure 6.2 such a subblock is no longer contiguous in memory. The way to describe this is by introducing a third parameter in addition to `M,N`: we let `LDA` be the 'leading dimension of `A`', that is, the allocated first dimension of the surrounding array. This is illustrated in figure 6.3. To pass the subblock to a routine, you would specify it as

```
call routine( A(3,2), /* M= */ 2, /* N= */ 3, /* LDA= */ Mbig, ... )
```

6.3 Performance issues

The collection of BLAS and LAPACK routines are a *de facto* standard: the API is fixed, but the implementation is not. You can find reference implementations on the *netlib* website (`netlib.org`), but these will be very low in performance.

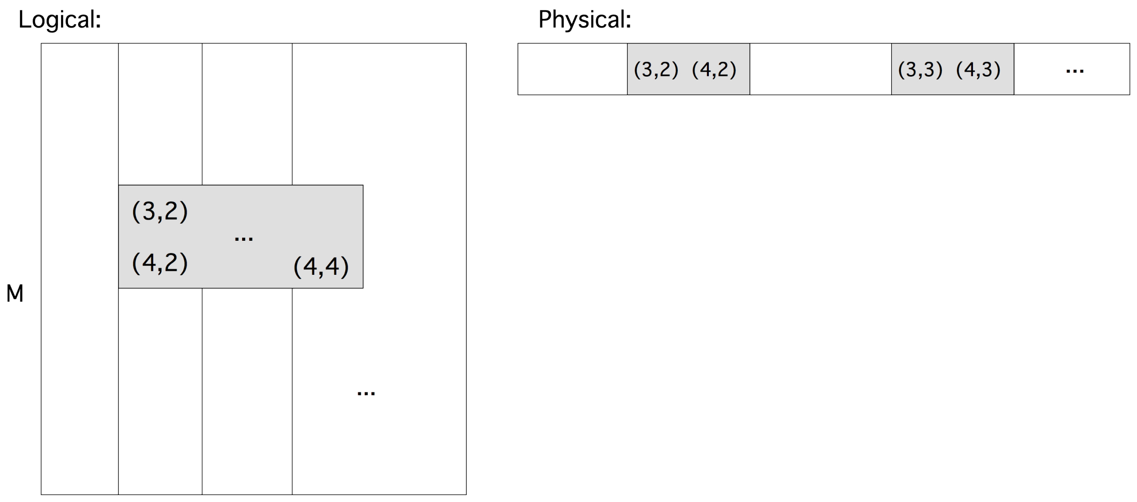

Figure 6.2: A subblock out of a larger matrix.

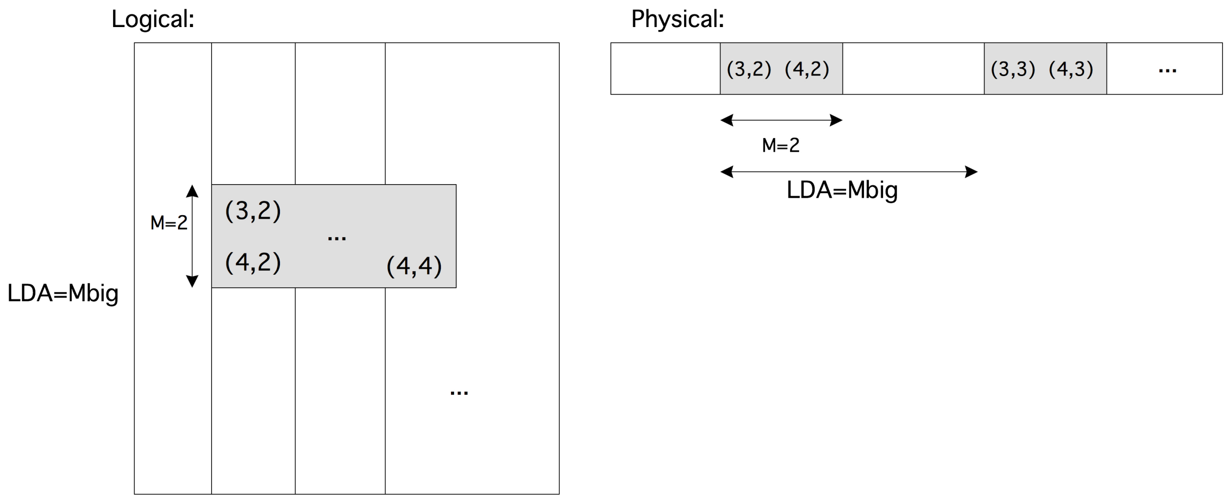

Figure 6.3: A subblock out of a larger matrix, using `LDA`.

On the other hand, many LAPACK routines can be based on the matrix-matrix product (BLAS routine `gemm`), which you saw in section *HPC book, section 7.4.1* has the potential for a substantial fraction of peak performance. To achieve this, you should use an optimized version, such as

- *MKL*, the Intel math-kernel library;
- OpenBlas (`http://www.openblas.net/`), an open source version of the original *Goto BLAS*; or
- *blis* (`https://code.google.com/p/blis/`), a BLAS replacement and extension project.

6.4 Some simple examples

Let's look at some simple examples.

The routine `xscal` scales a vector in place.

```
! Fortran
subroutine dscal(integer N, double precision DA,
    double precision, dimension(*) DX, integer INCX )
// C
void cblas_dscal (const MKL_INT n, const double a,
    double *x, const MKL_INT incx);
```

A simple example:

```
// /Users/eijkhout/Current/istc/scientific-computing-private/tutorials/blas/example1.F90
do i=1,n
   xarray(i) = 1.d0
end do
call dscal(n,scale,xarray,1)
do i=1,n
   if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
end do
```

The same in C:

```
// /Users/eijkhout/Current/istc/scientific-computing-private/tutorials/blas/example1c.cpp
xarray = new double[n]; yarray = new double[n];

for (int i=0; i<n; i++)
  xarray[i] = 1.;
cblas_dscal(n,scale,xarray,1);
for (int i=0; i<n; i++)
  if (!assert_equal( xarray[i],scale ))
    printf("Error in index %d",i);
```

Many routines have an increment parameter. For `xscale` that's the final parameter:

```
// /Users/eijkhout/Current/istc/scientific-computing-private/tutorials/blas/example2.F90
integer :: inc=2
  /* ... */
call dscal(n/inc,scale,xarray,inc)
do i=1,n
   if (mod(i,inc)==1) then
      if (.not.assert_equal( xarray(i),scale )) print *,"Error in index",i
   else
      if (.not.assert_equal( xarray(i),1.d0 )) print *,"Error in index",i
   end if
end do
```

The matrix-vector product `xgemv` computes $y \leftarrow \alpha Ax + \beta y$, rather than $y \leftarrow Ax$. The specification of the matrix takes the `M,N` size parameters, and a character argument `'N'` to indicate that the matrix is not transposed. Both of the vectors have an increment argument.

```
subroutine dgemv(character TRANS,
    integer M,integer N,
    double precision ALPHA,
    double precision, dimension(lda,*) A,integer LDA,
    double precision, dimension(*) X,integer INCX,
    double precision BETA,double precision, dimension(*) Y,integer INCY
    )
```

An example of the use of this routine:

```
// /Users/eijkhout/Current/istc/scientific-computing-private/tutorials/blas/example3.F90
do j=1,n
   xarray(j) = 1.d0
   do i=1,m
      matrix(i,j) = 1.d0
   end do
end do

alpha = 1.d0; beta = 0.d0
call dgemv( 'N',M,N, alpha,matrix,M, xarray,1, beta,yarray,1)
do i=1,m
   if (.not.assert_equal( yarray(i),dble(n) )) &
        print *,"Error in index",i,":",yarray(i)
end do
```

The same example in C has an extra parameter to indicate whether the matrix is stored in row or column major storage:

```
// /Users/eijkhout/Current/istc/scientific-computing-private/tutorials/blas/example3c.cpp
  for (int j=0; j<n; j++) {
    xarray[j] = 1.;
    for (int i=0; i<m; i++)
      matrix[ i+j*m ] = 1.;
  }

  alpha = 1.; beta = 0.;
  cblas_dgemv(CblasColMajor,
              CblasNoTrans,m,n, alpha,matrix,m, xarray,1, beta,yarray,1);

  for (int i=0; i<m; i++)
    if (!assert_equal( yarray[i],(double)n ))
      printf("Error in index %d",i);
```