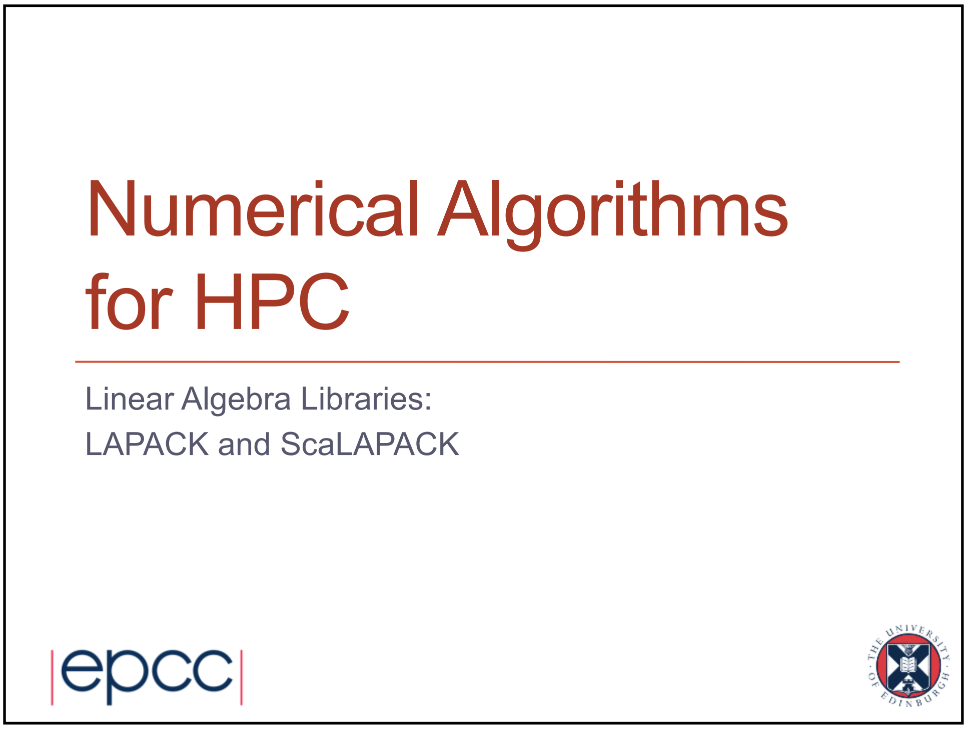
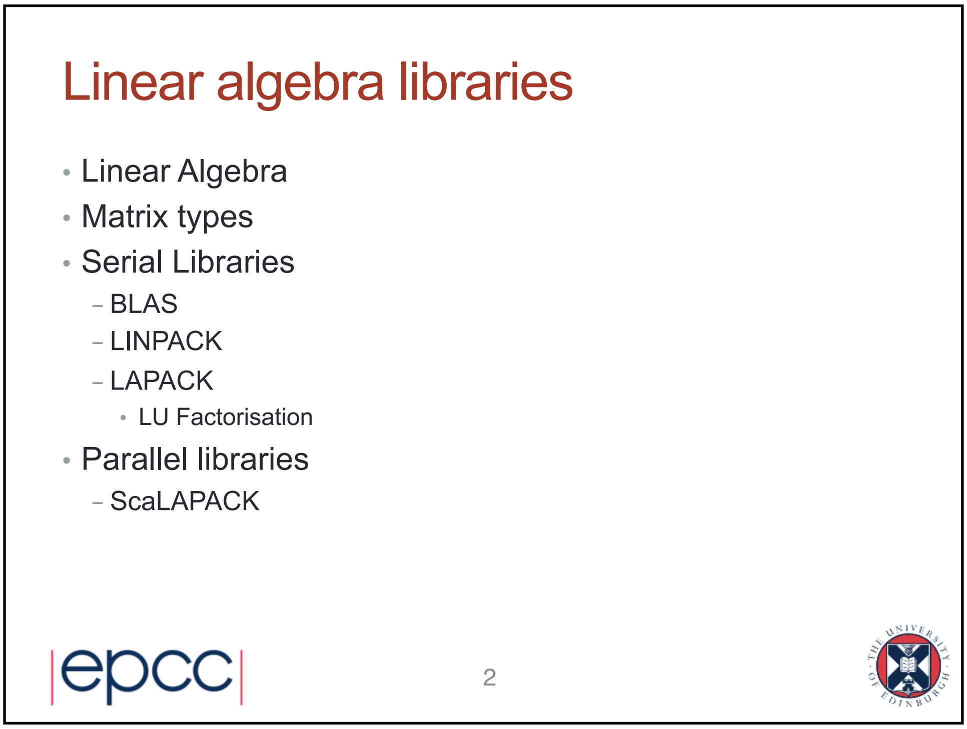


# Numerical Algorithms for HPC

Linear Algebra Libraries:
LAPACK and ScaLAPACK

1



## Linear algebra libraries

- Linear Algebra
- Matrix types
- Serial Libraries
  - BLAS
  - LINPACK
  - LAPACK
    - LU Factorisation
- Parallel libraries
  - ScaLAPACK

2

2

## Linear algebra libraries

- Linear Algebra is a well constrained problem
  - can define a small set of common operations
  - implement them robustly and efficiently in a library
  - mainly designed to be called from Fortran (see later ...)
- Often seen as the most important HPC library
  - e.g. LINPACK benchmark

TOP 500

The List.

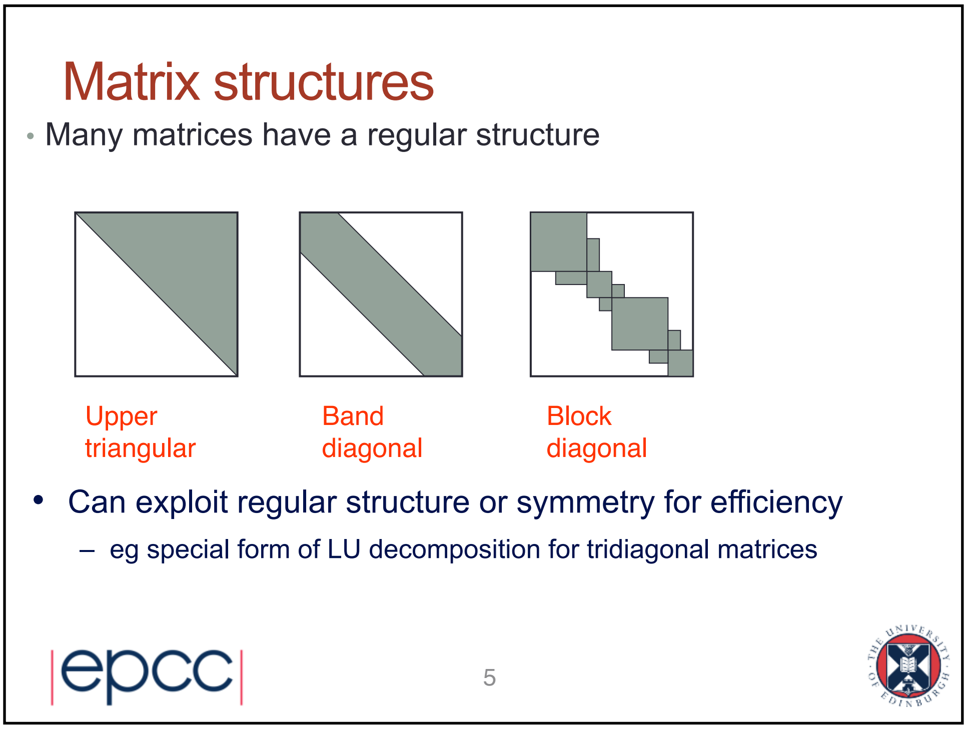
- Linear algebra is unusually efficient
  - LU decomposition has O($N^3$) operations for O($N^2$) memory loads

## Matrix types

- Matrices generally classified as either *sparse* or *dense*
  - We will deal with sparse matrices later
- Rectangular matrices
  - correspond to different number of equations from unknowns
  - system can be either under- or over-determined
- Matrices may have symmetry about the diagonal:
  - Symmetric (real matrices): $a_{ij} = a_{ji} \longrightarrow A^T = A$
  - Hermitian (complex matrices): $a_{ij}^* = a_{ji} \longrightarrow A^{T*} = A^\dagger = A$

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2+i \\ 2-i & 3 \end{pmatrix}$$

Symmetric Hermitian



## Matrix structures

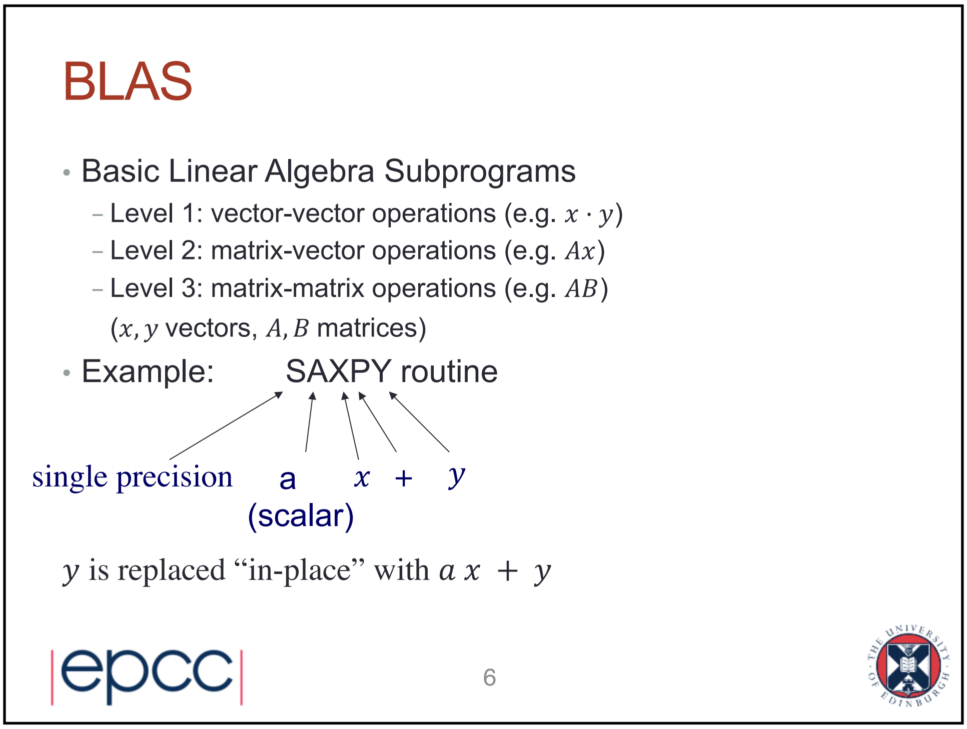
- Many matrices have a regular structure

Upper triangular
Band diagonal
Block diagonal

- Can exploit regular structure or symmetry for efficiency
  - eg special form of LU decomposition for tridiagonal matrices

5

5



## BLAS

- Basic Linear Algebra Subprograms
  - Level 1: vector-vector operations (e.g. $x \cdot y$)
  - Level 2: matrix-vector operations (e.g. $Ax$)
  - Level 3: matrix-matrix operations (e.g. $AB$)

  ($x, y$ vectors, $A, B$ matrices)

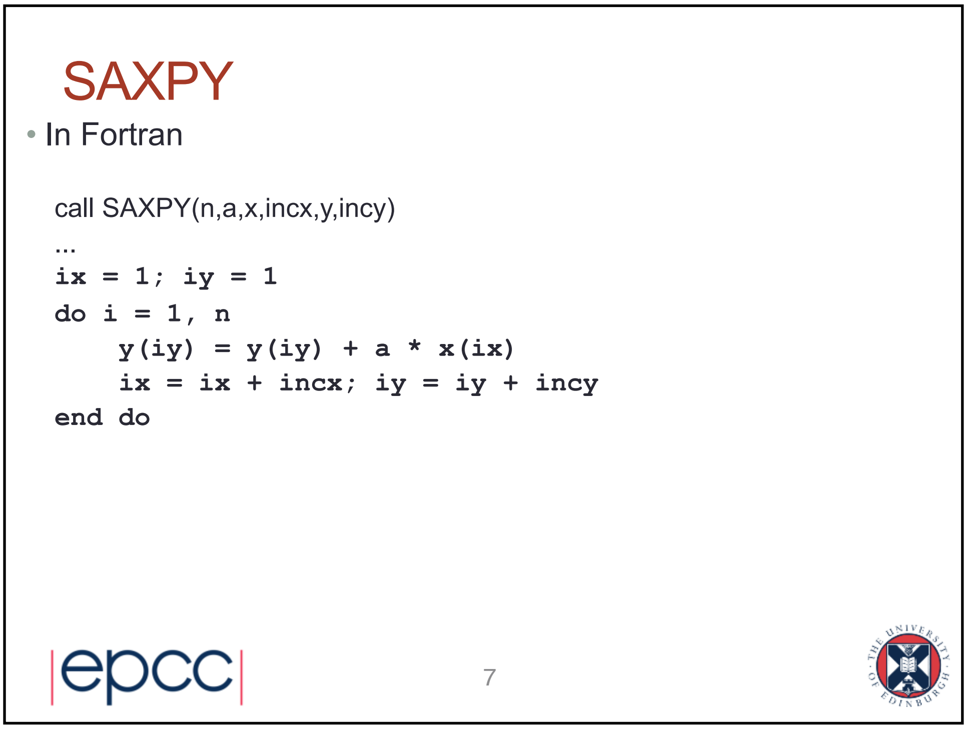
- Example: SAXPY routine

single precision
a
(scalar)
$x$
+
$y$

$y$ is replaced “in-place” with $a\,x + y$

6

6

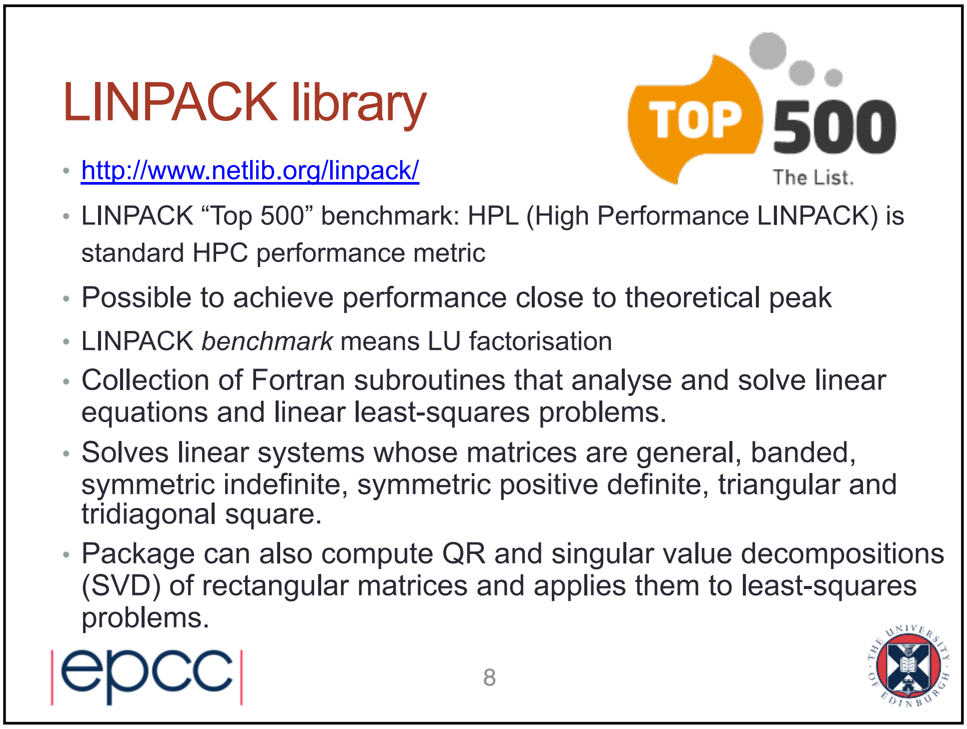


## SAXPY

- In Fortran

```
call SAXPY(n,a,x,incx,y,incy)
...
ix = 1; iy = 1
do i = 1, n
    y(iy) = y(iy) + a * x(ix)
    ix = ix + incx; iy = iy + incy
end do
```

7



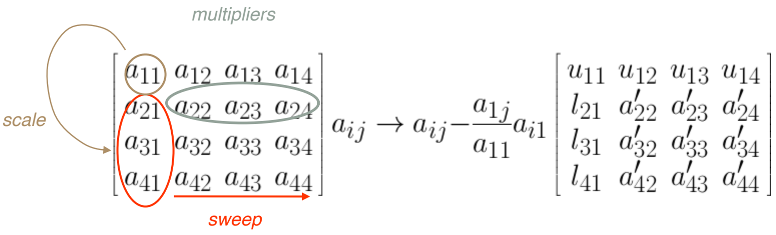
## LINPACK library

- http://www.netlib.org/linpack/
- LINPACK “Top 500” benchmark: HPL (High Performance LINPACK) is standard HPC performance metric
- Possible to achieve performance close to theoretical peak
- LINPACK *benchmark* means LU factorisation
- Collection of Fortran subroutines that analyse and solve linear equations and linear least-squares problems.
- Solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular and tridiagonal square.
- Package can also compute QR and singular value decompositions (SVD) of rectangular matrices and applies them to least-squares problems.

8

## Why is performance so high? LU factorisation

- Look at step in LU decomposition
  - consider an in-place decomposition
  - for each column $k$
    - scale the sub-diagonal column below $a_{kk}$ by $1/a_{kk}$
    - for all columns *j* to the right, subtract $a_{kj}$ times above
  - e.g. for $k = 1$



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad a_{ij} \to a_{ij} - \frac{a_{1j}}{a_{11}} a_{i1} \quad \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ l_{21} & a'_{22} & a'_{23} & a'_{24} \\ l_{31} & a'_{32} & a'_{33} & a'_{34} \\ l_{41} & a'_{42} & a'_{43} & a'_{44} \end{bmatrix}$$

- These are vector operations over columns
  - scaling a vector $x$ by *a* (a scalar variable): $x_i \to a x_i$
  - updating a second vector $y$ by the above: $y_i \to a x_i + y_i$

epcc

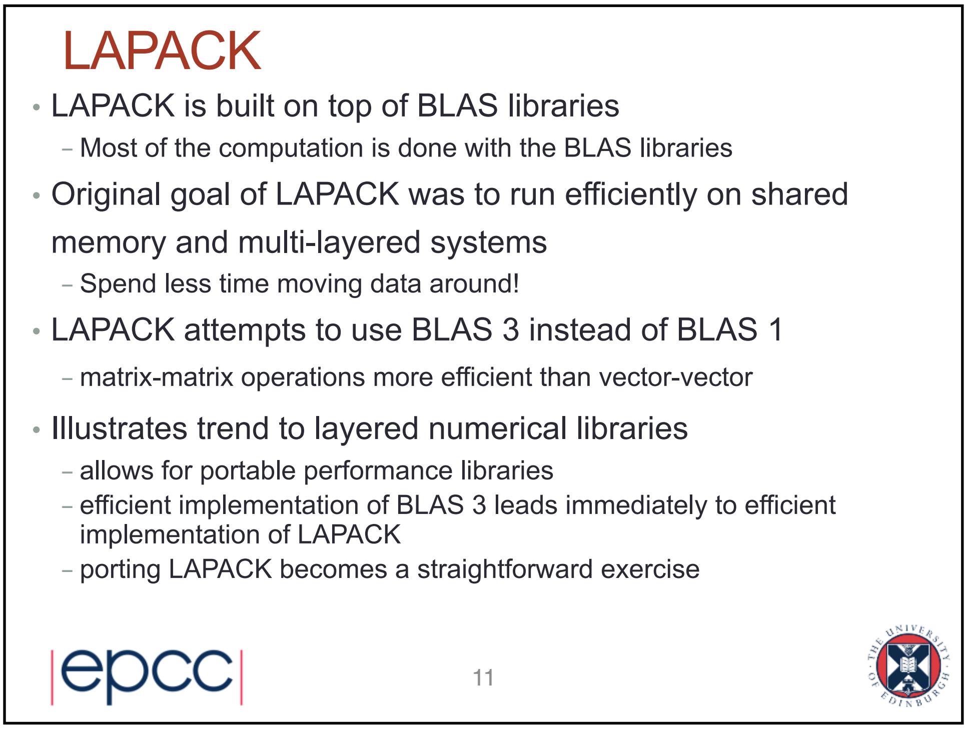
## LINPACK implementation

- Based around level 1 BLAS
  - e.g. look at LU decomposition

```
t = -1.0e0/a(k,k)
call SSCAL(n-k,t,a(k+1,k),1)
...
do j = k+1,n
   call SAXPY(n-k,-a(k,j),a(k+1,k),1,a(k+1,j),1)
end do
```

- Very efficient on early HPC vector machines
  - not nearly so efficient on modern cache-based systems

epcc

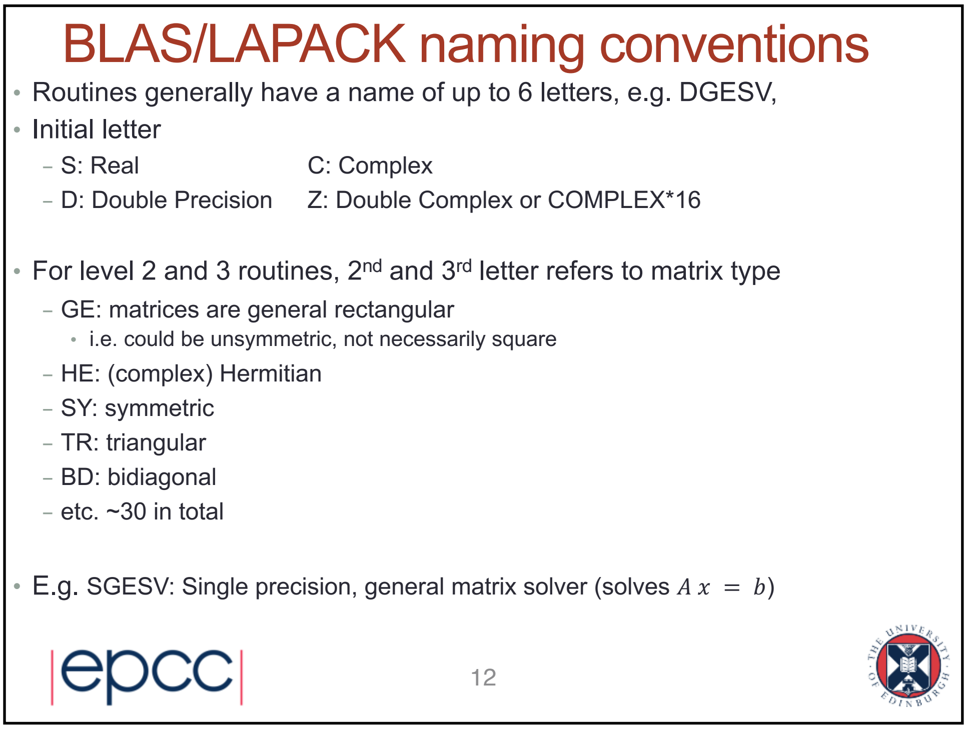


# LAPACK

- LAPACK is built on top of BLAS libraries
  - Most of the computation is done with the BLAS libraries
- Original goal of LAPACK was to run efficiently on shared memory and multi-layered systems
  - Spend less time moving data around!
- LAPACK attempts to use BLAS 3 instead of BLAS 1
  - matrix-matrix operations more efficient than vector-vector
- Illustrates trend to layered numerical libraries
  - allows for portable performance libraries
  - efficient implementation of BLAS 3 leads immediately to efficient implementation of LAPACK
  - porting LAPACK becomes a straightforward exercise

epcc

11



# BLAS/LAPACK naming conventions

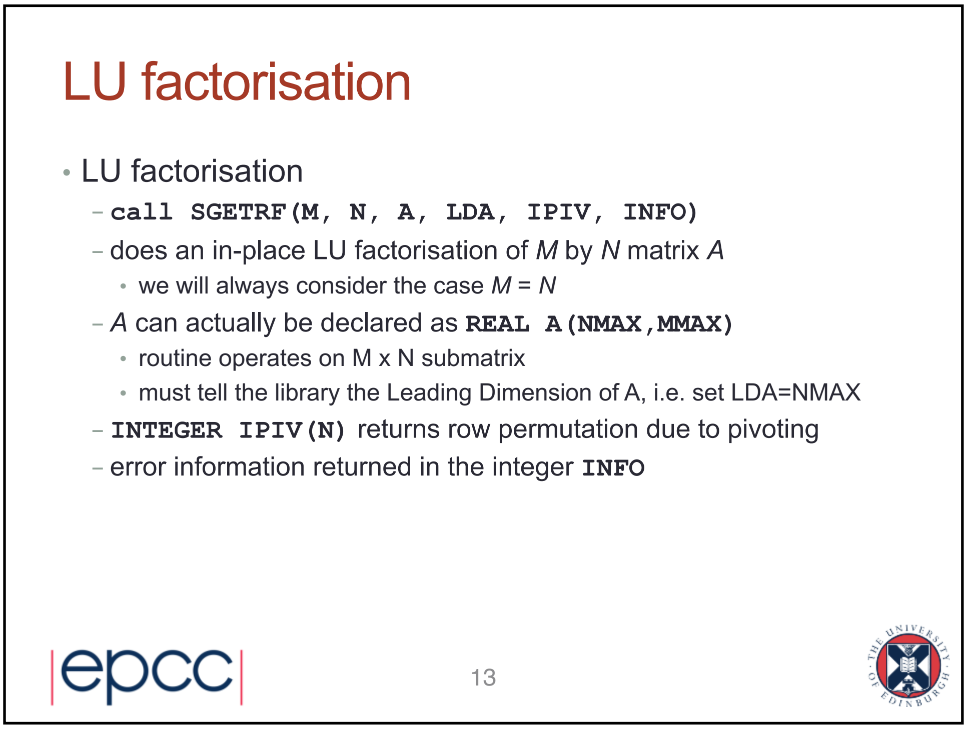
- Routines generally have a name of up to 6 letters, e.g. DGESV,
- Initial letter

| | |
|---|---|
| – S: Real | C: Complex |
| – D: Double Precision | Z: Double Complex or COMPLEX*16 |

- For level 2 and 3 routines, 2nd and 3rd letter refers to matrix type
  - GE: matrices are general rectangular
    - i.e. could be unsymmetric, not necessarily square
  - HE: (complex) Hermitian
  - SY: symmetric
  - TR: triangular
  - BD: bidiagonal
  - etc. ~30 in total
- E.g. SGESV: Single precision, general matrix solver (solves $A\,x\; = \; b$)

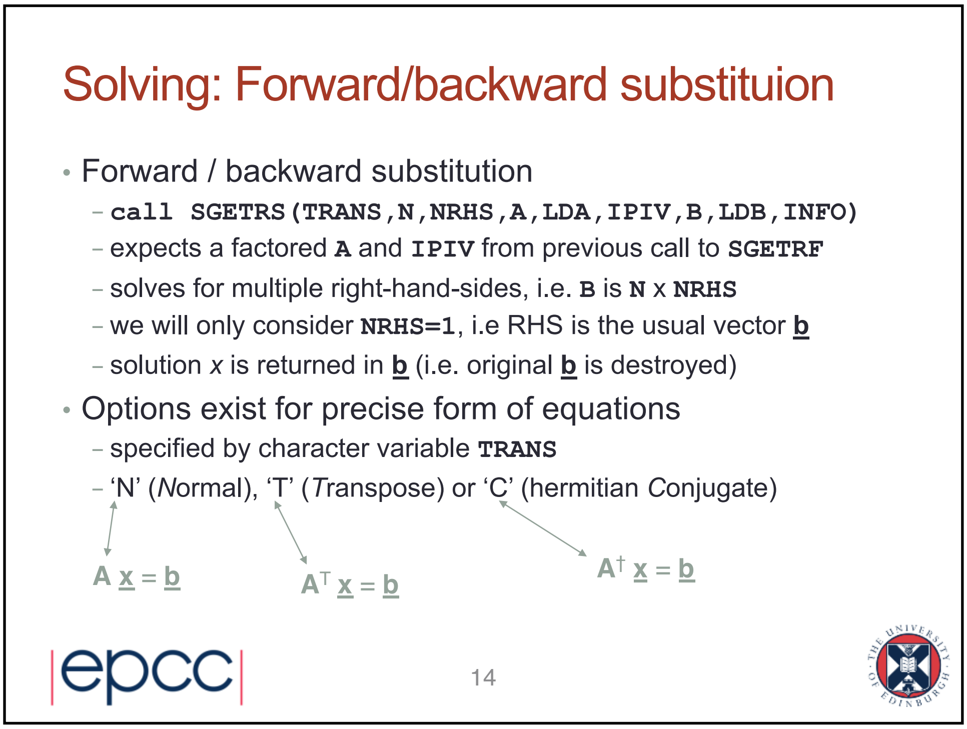
epcc

12



# LU factorisation

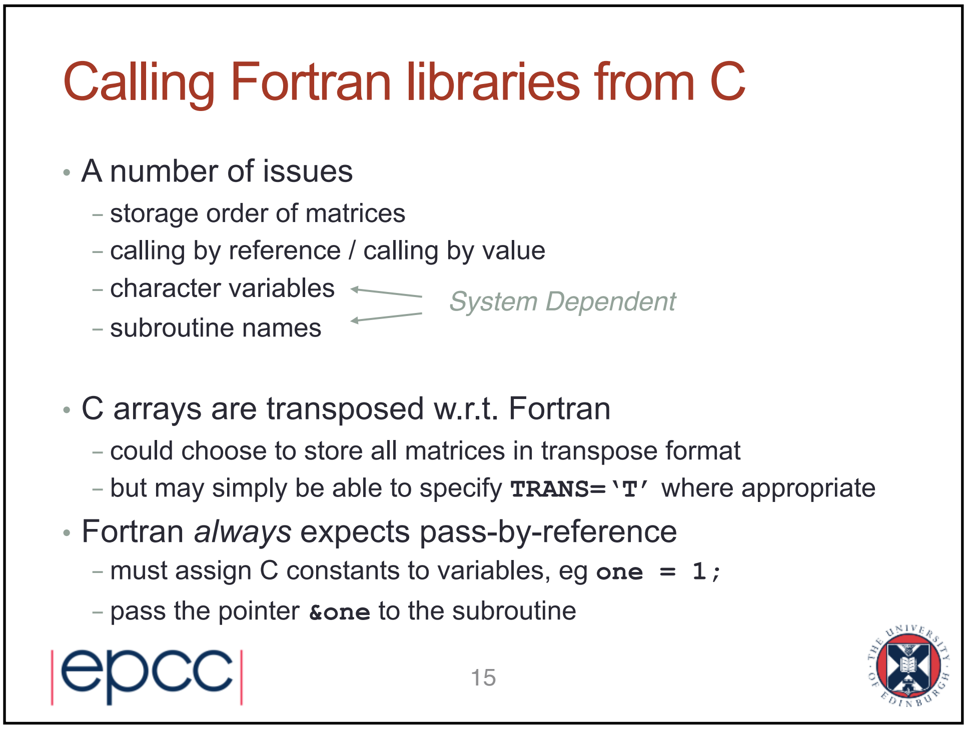
- LU factorisation
  - `call SGETRF(M, N, A, LDA, IPIV, INFO)`
  - does an in-place LU factorisation of *M* by *N* matrix *A*
    - we will always consider the case *M* = *N*
  - *A* can actually be declared as **`REAL A(NMAX,MMAX)`**
    - routine operates on M x N submatrix
    - must tell the library the Leading Dimension of A, i.e. set LDA=NMAX
  - **`INTEGER IPIV(N)`** returns row permutation due to pivoting
  - error information returned in the integer **`INFO`**



# Solving: Forward/backward substituion

- Forward / backward substitution
  - `call SGETRS(TRANS,N,NRHS,A,LDA,IPIV,B,LDB,INFO)`
  - expects a factored **`A`** and **`IPIV`** from previous call to **`SGETRF`**
  - solves for multiple right-hand-sides, i.e. **`B`** is **`N`** x **`NRHS`**
  - we will only consider **`NRHS=1`**, i.e RHS is the usual vector **b**
  - solution *x* is returned in **b** (i.e. original **b** is destroyed)
- Options exist for precise form of equations
  - specified by character variable **`TRANS`**
  - 'N' (*N*ormal), 'T' (*T*ranspose) or 'C' (hermitian *C*onjugate)

$A \underline{x} = \underline{b}$ $\quad A^{T} \underline{x} = \underline{b}$ $\quad A^{\dagger} \underline{x} = \underline{b}$



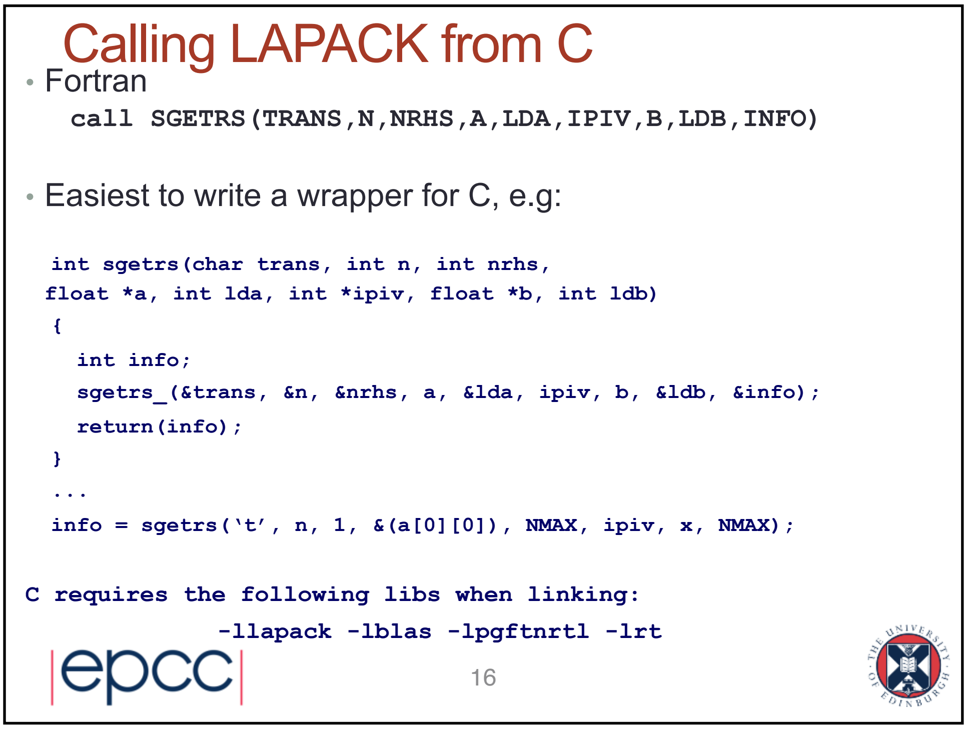
# Calling Fortran libraries from C

- A number of issues
  - storage order of matrices
  - calling by reference / calling by value
  - character variables
  - subroutine names

*System Dependent*

- C arrays are transposed w.r.t. Fortran
  - could choose to store all matrices in transpose format
  - but may simply be able to specify **`TRANS='T'`** where appropriate
- Fortran *always* expects pass-by-reference
  - must assign C constants to variables, eg **`one = 1;`**
  - pass the pointer **`&one`** to the subroutine

15



# Calling LAPACK from C

- Fortran

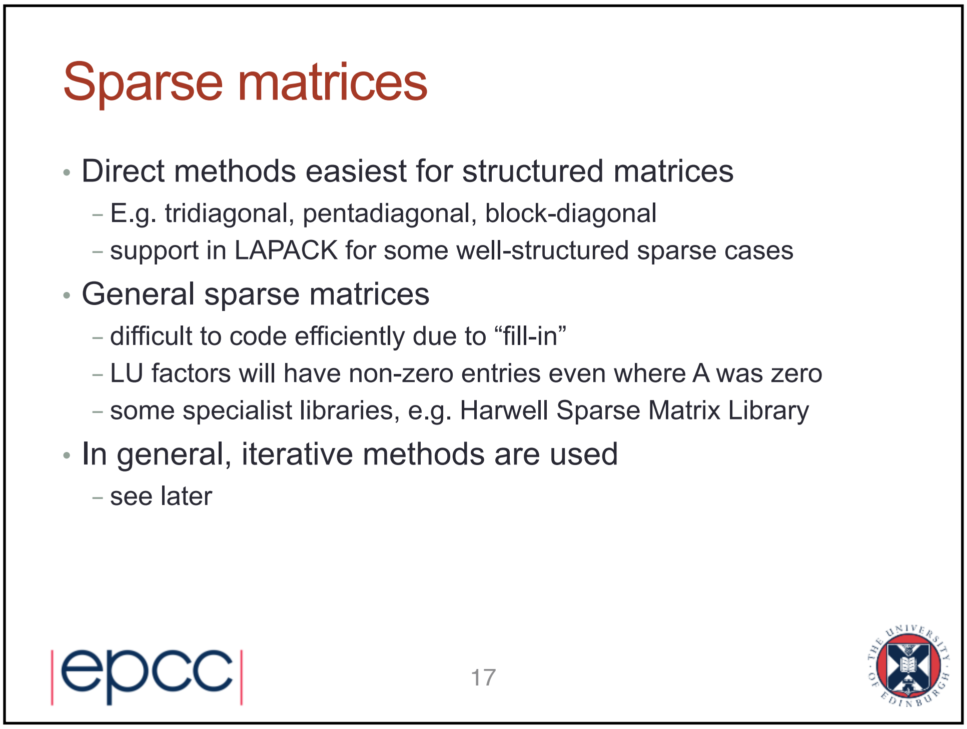
```
call SGETRS(TRANS,N,NRHS,A,LDA,IPIV,B,LDB,INFO)
```

- Easiest to write a wrapper for C, e.g:

```
 int sgetrs(char trans, int n, int nrhs,
float *a, int lda, int *ipiv, float *b, int ldb)
 {
   int info;
   sgetrs_(&trans, &n, &nrhs, a, &lda, ipiv, b, &ldb, &info);
   return(info);
 }
 ...
 info = sgetrs('t', n, 1, &(a[0][0]), NMAX, ipiv, x, NMAX);
```

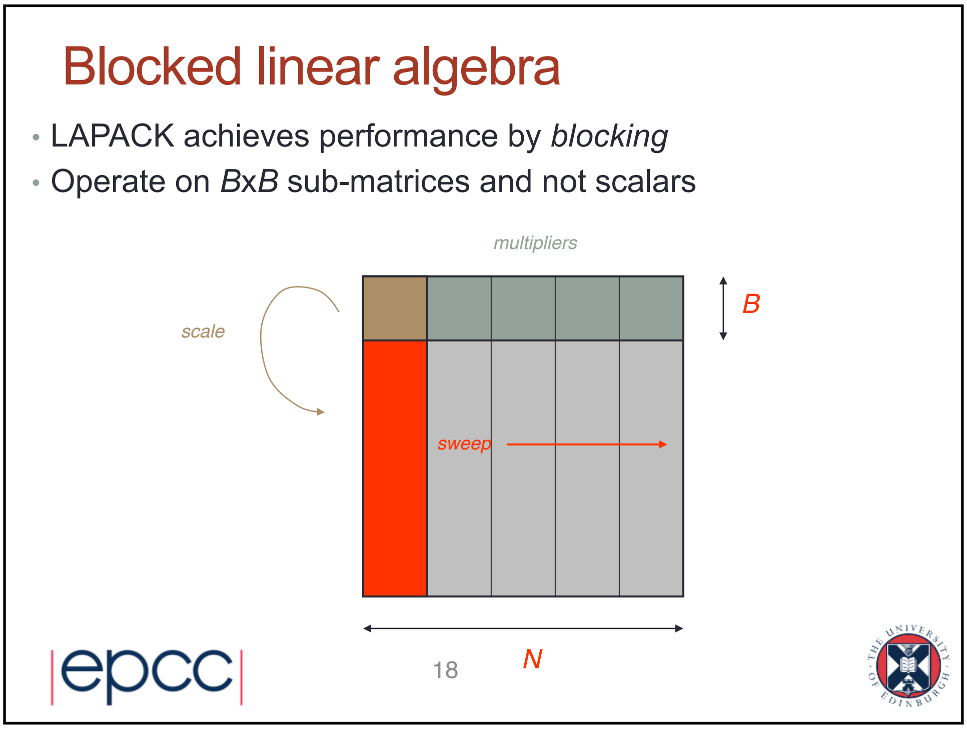
```
C requires the following libs when linking:
                  -llapack -lblas -lpgftnrtl -lrt
```

16



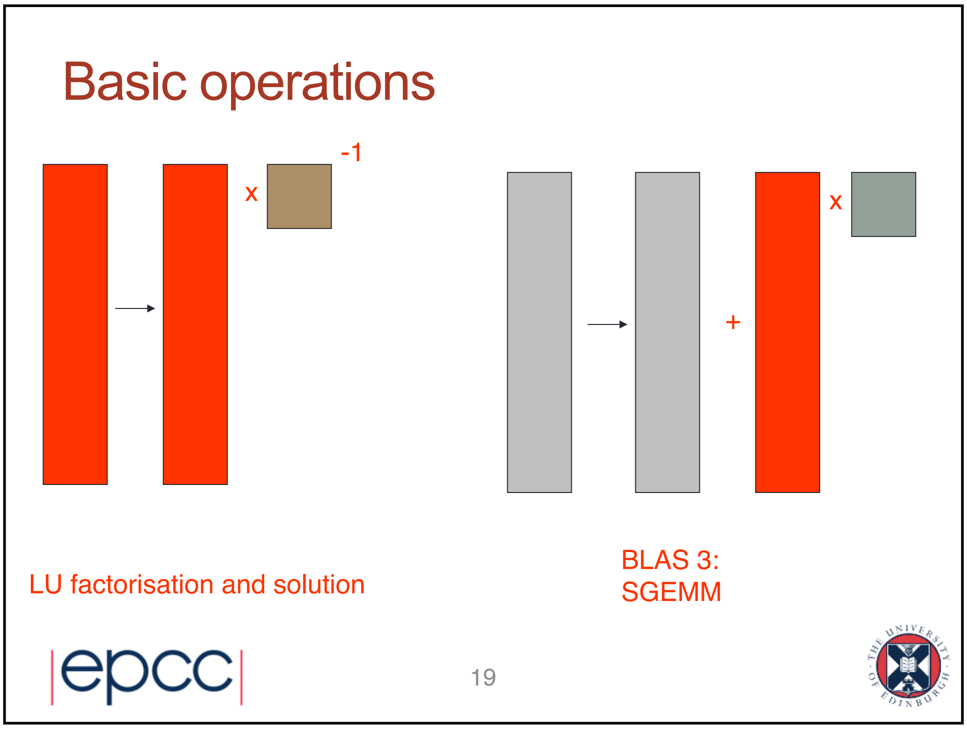
# Sparse matrices

- Direct methods easiest for structured matrices
  - E.g. tridiagonal, pentadiagonal, block-diagonal
  - support in LAPACK for some well-structured sparse cases
- General sparse matrices
  - difficult to code efficiently due to “fill-in”
  - LU factors will have non-zero entries even where A was zero
  - some specialist libraries, e.g. Harwell Sparse Matrix Library
- In general, iterative methods are used
  - see later



# Blocked linear algebra

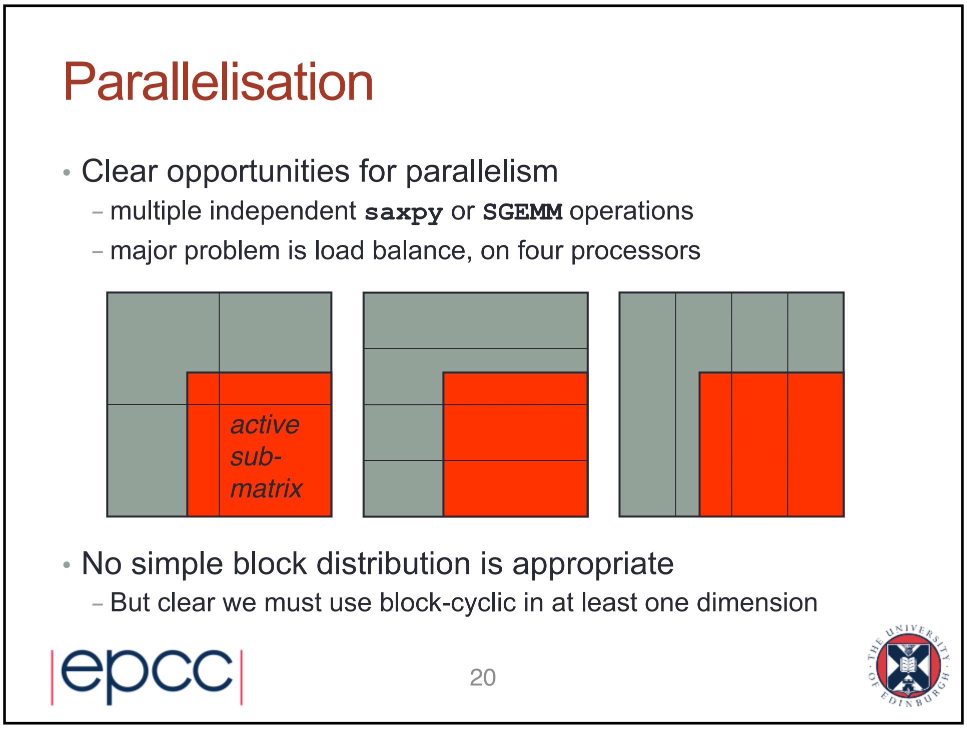
- LAPACK achieves performance by *blocking*
- Operate on *BxB* sub-matrices and not scalars



## Basic operations

LU factorisation and solution

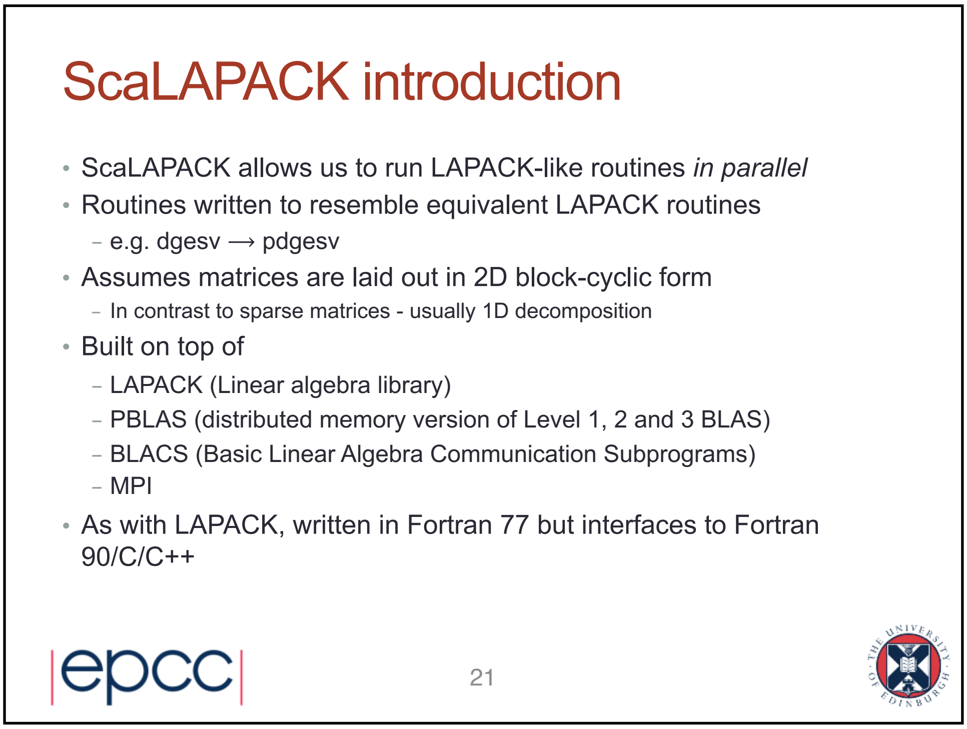
BLAS 3:
SGEMM



## Parallelisation

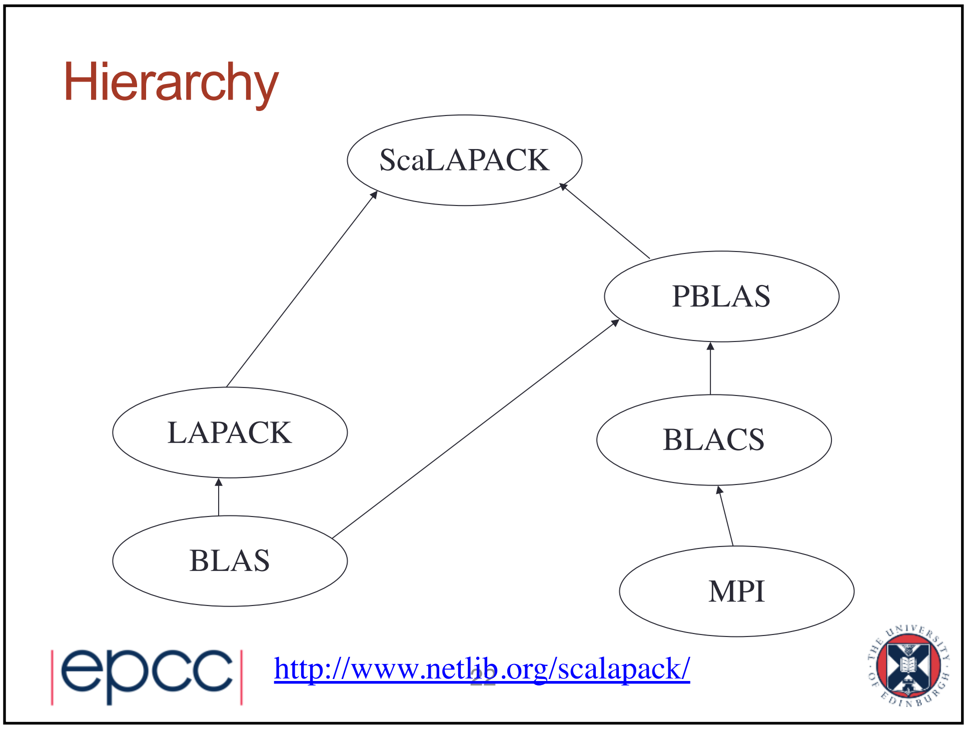
- Clear opportunities for parallelism
  - multiple independent `saxpy` or `SGEMM` operations
  - major problem is load balance, on four processors

- No simple block distribution is appropriate
  - But clear we must use block-cyclic in at least one dimension



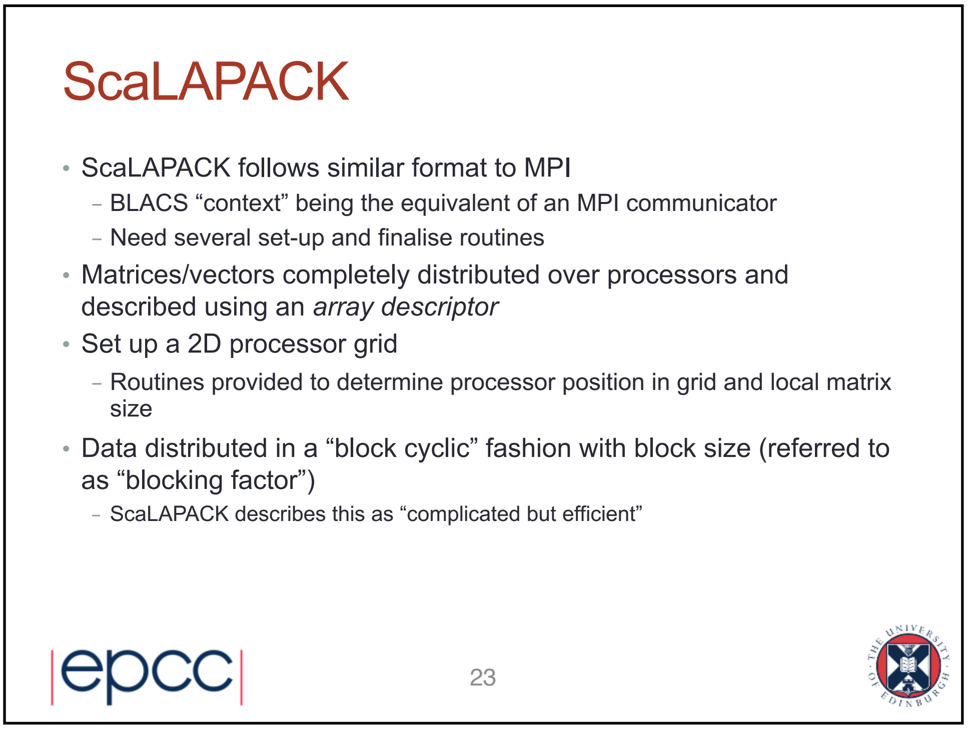
# ScaLAPACK introduction

- ScaLAPACK allows us to run LAPACK-like routines *in parallel*
- Routines written to resemble equivalent LAPACK routines
  - e.g. dgesv ⟶ pdgesv
- Assumes matrices are laid out in 2D block-cyclic form
  - In contrast to sparse matrices - usually 1D decomposition
- Built on top of
  - LAPACK (Linear algebra library)
  - PBLAS (distributed memory version of Level 1, 2 and 3 BLAS)
  - BLACS (Basic Linear Algebra Communication Subprograms)
  - MPI
- As with LAPACK, written in Fortran 77 but interfaces to Fortran 90/C/C++



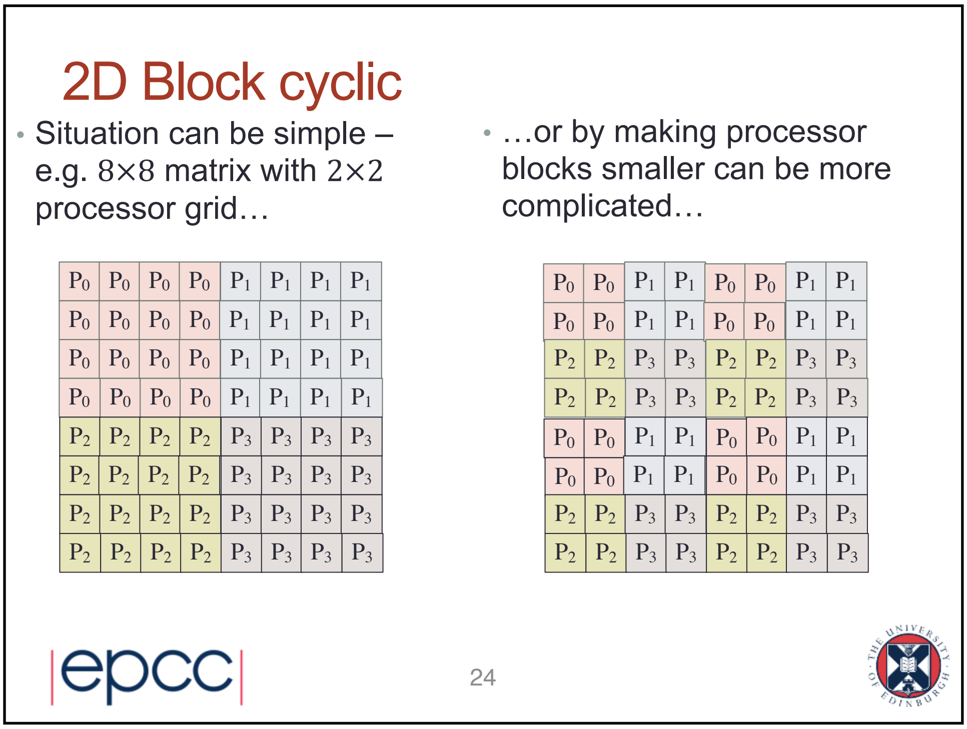
# Hierarchy

http://www.netlib.org/scalapack/



# ScaLAPACK

- ScaLAPACK follows similar format to MPI
  - BLACS “context” being the equivalent of an MPI communicator
  - Need several set-up and finalise routines
- Matrices/vectors completely distributed over processors and described using an *array descriptor*
- Set up a 2D processor grid
  - Routines provided to determine processor position in grid and local matrix size
- Data distributed in a “block cyclic” fashion with block size (referred to as “blocking factor”)
  - ScaLAPACK describes this as “complicated but efficient”



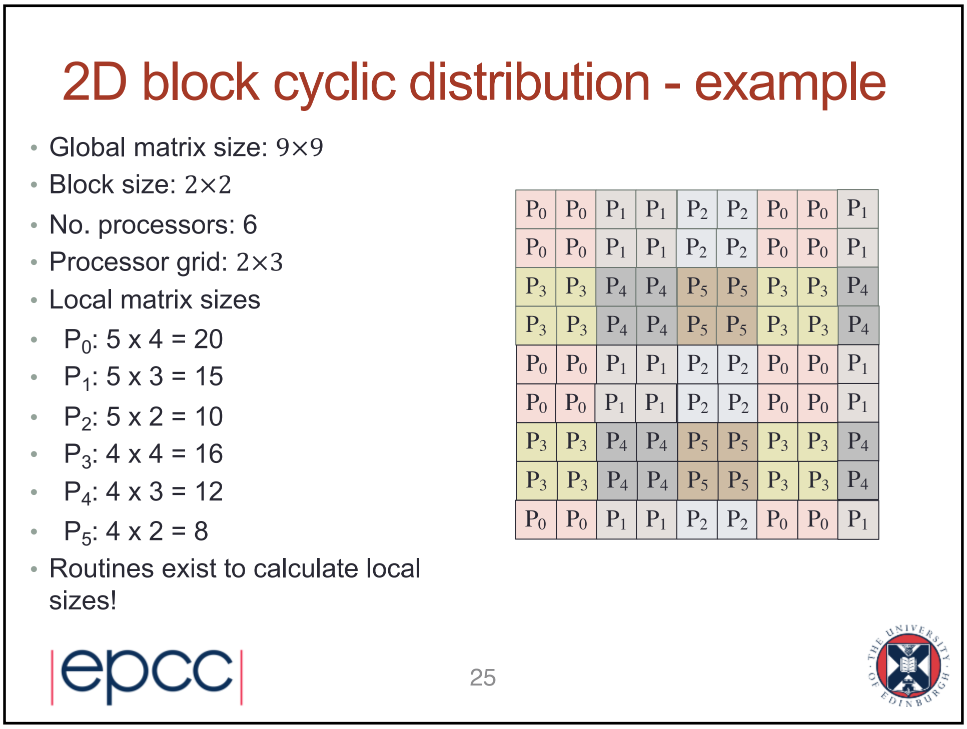
# 2D Block cyclic

- Situation can be simple – e.g. 8×8 matrix with 2×2 processor grid…

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $P_0$ | $P_0$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
| $P_0$ | $P_0$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
| $P_0$ | $P_0$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
| $P_0$ | $P_0$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
| $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_3$ | $P_3$ |
| $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_3$ | $P_3$ |
| $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_3$ | $P_3$ |
| $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_3$ | $P_3$ |

- …or by making processor blocks smaller can be more complicated…

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ |
| $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ |
| $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_0$ | $P_0$ | $P_1$ | $P_1$ |
| $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ |
| $P_2$ | $P_2$ | $P_3$ | $P_3$ | $P_2$ | $P_2$ | $P_3$ | $P_3$ |



## 2D block cyclic distribution - example

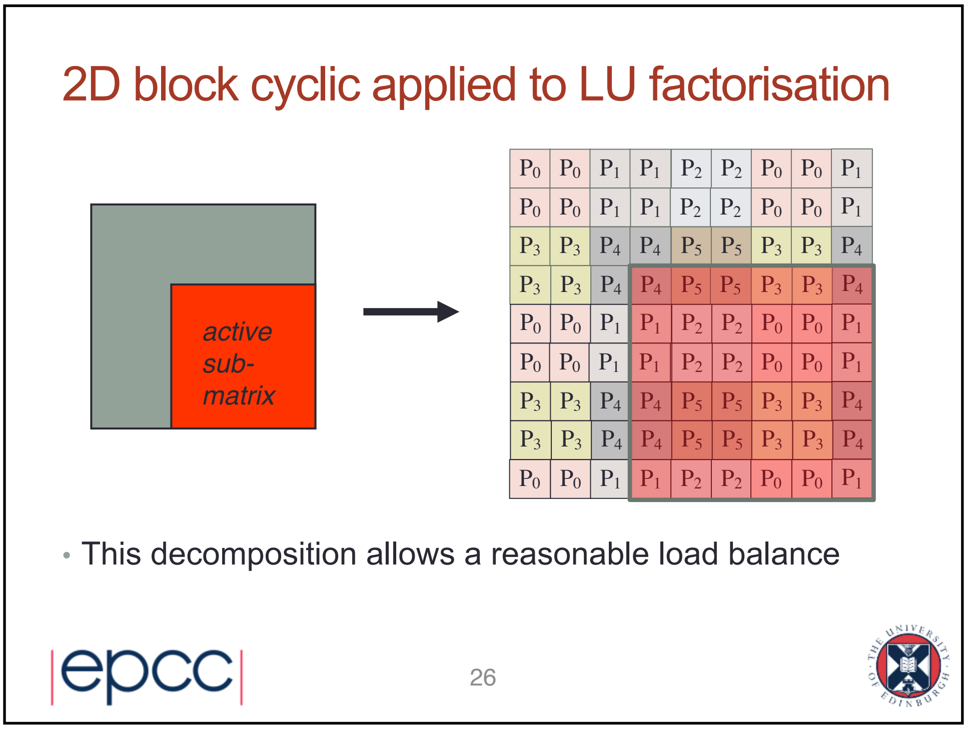
- Global matrix size: $9\times 9$
- Block size: $2\times 2$
- No. processors: 6
- Processor grid: $2\times 3$
- Local matrix sizes
- $P_0$: 5 x 4 = 20
- $P_1$: 5 x 3 = 15
- $P_2$: 5 x 2 = 10
- $P_3$: 4 x 4 = 16
- $P_4$: 4 x 3 = 12
- $P_5$: 4 x 2 = 8
- Routines exist to calculate local sizes!

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |

epcc

25

25



## 2D block cyclic applied to LU factorisation

*active sub-matrix*

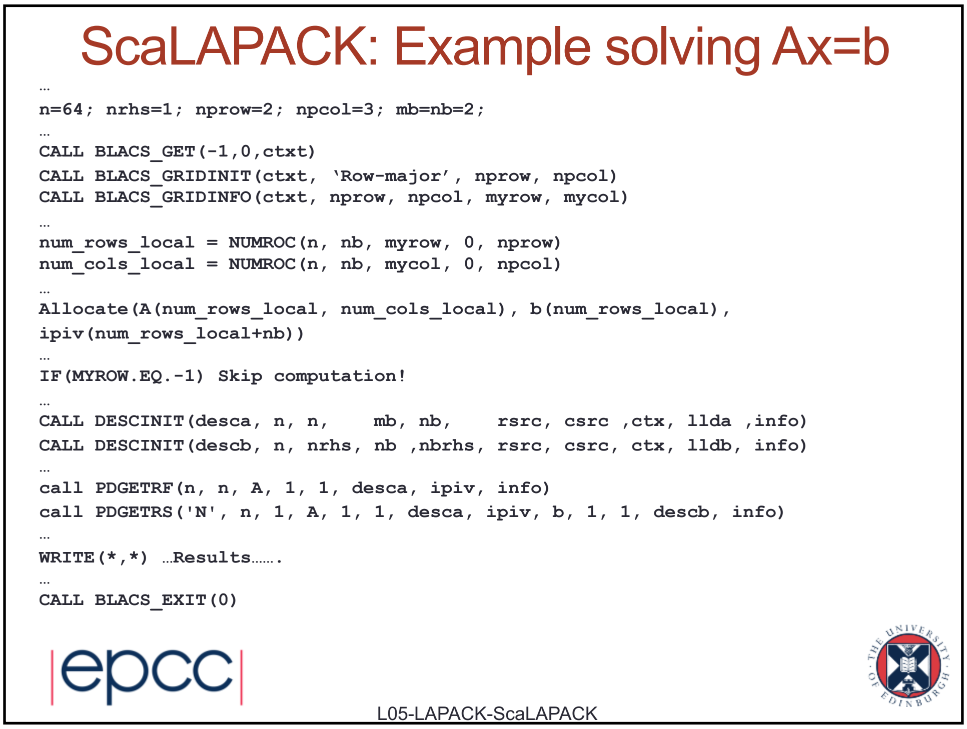
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_3$ | $P_3$ | $P_4$ | $P_4$ | $P_5$ | $P_5$ | $P_3$ | $P_3$ | $P_4$ |
| $P_0$ | $P_0$ | $P_1$ | $P_1$ | $P_2$ | $P_2$ | $P_0$ | $P_0$ | $P_1$ |

- This decomposition allows a reasonable load balance

epcc

26

26



# ScaLAPACK: Example solving Ax=b

```
…
n=64; nrhs=1; nprow=2; npcol=3; mb=nb=2;
…
CALL BLACS_GET(-1,0,ctxt)
CALL BLACS_GRIDINIT(ctxt, ‘Row-major’, nprow, npcol)
CALL BLACS_GRIDINFO(ctxt, nprow, npcol, myrow, mycol)
…
num_rows_local = NUMROC(n, nb, myrow, 0, nprow)
num_cols_local = NUMROC(n, nb, mycol, 0, npcol)
…
Allocate(A(num_rows_local, num_cols_local), b(num_rows_local),
ipiv(num_rows_local+nb))
…
IF(MYROW.EQ.-1) Skip computation!
…
CALL DESCINIT(desca, n, n,    mb, nb,    rsrc, csrc ,ctx, llda ,info)
CALL DESCINIT(descb, n, nrhs, nb ,nbrhs, rsrc, csrc, ctx, lldb, info)
…
call PDGETRF(n, n, A, 1, 1, desca, ipiv, info)
call PDGETRS('N', n, 1, A, 1, 1, desca, ipiv, b, 1, 1, descb, info)
…
WRITE(*,*) …Results…….
…
CALL BLACS_EXIT(0)
```

L05-LAPACK-ScaLAPACK

27