HPCSE II

Dense linear algebra

Basic Linear Algebra Subprograms (BLAS)

- BLAS is the de-facto standard API for any dense vector and matrix operations, first published in 1979
- A reference implementation is available on netlib.org:
  - http://www.netlib.org/blas/ for the original Fortran version
  - http://www.netlib.org/clapack/ for an f2c translated C version
- BLAS is the building block of many other libraries and programs. These libraries rely on an optimized BLAS library for optimal performance
  - LAPACK and LINPACK
  - NAG (commercial)
  - IMSL (commercial)
  - Matlab
  - Python (numpy and scipy)

Optimized BLAS implementations

- Don't get the reference implementation but an optimized one.
  - ATLAS (free) http://www.netlib.org/atlas/, self-tuned BLAS, included with many Linux distributions
  - GOTO BLAS http://www.tacc.utexas.edu/tacc-projects/gotoblas2/ hand-optimized by Kazushige Goto before he was hired by Microsoft
- Optimized versions exist from many hardware vendors:
  - Apple: included with MacOS X in the Accelerate framework (-framework Accelerate)
  - IBM: part of the ESSL (Engineering and Scientific Subroutine Library)
  - Cray: part of libsci library
  - NEC: part of the PDLIB/SX library
- And from CPU manufacturers
  - Intel MKL library http://software.intel.com/en-us/intel-mkl/ , available on IDES
  - AMD ACML library http://developer.amd.com/tools/cpu/acml/pages/default.aspx
- Homework: get the fastest BLAS for all your computers

BLAS levels 1, 2 and 3

- The BLAS functions are split into three groups
- BLAS level 1
  - scalar and vector operations
  - scale as $O(1)$ *or* $O(N)$
- BLAS level 2
  - matrix-vector operations
  - scale as $O(N^2)$
- BLAS level 3
  - matrix-matrix operations
  - scale worse than $O(N^2)$, often $O(N^3)$

Calling BLAS functions

- BLAS is a Fortran library. It can be called from any language but you have to learn some facts about Fortran and calling Fortran functions.

- Function names and arguments:
  - The function names are all lowercase independent of what is written in (case-insensitive) Fortran code
  - Function names on most machines add a trailing _ compared to C/C++ functions.
  - Parameter types are not mangled into the function name:
    use **extern "C"** in the function declaration
  - all arguments are passed by address (or equivalently reference in C++). The best convention is to
    - pass scalar arguments by reference
    - pass C-style arrays as pointers
  - Be careful about how integer types relate. This can depend on compiler options. Typically a Fortran integer is a C/C++ int, but it can be a long.

Example: DDOT

- The Fortran DDOT function

```
      DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
*
*     DDOT forms the dot product of two vectors.
*     uses unrolled loops for increments equal to one.
*
```

- has the following C++ prototype

```
extern “C” double ddot_(int& n, double *x, int& incx, double *y, int& incy);
```

- and can easily be called.

```
int main()
{
  std::vector<double> x(10, 1.); // intialize a vector with ten 1s
  std::vector<double> y(10, 2.); // intialize a vector with ten 2s

  // calculate the inner product
  int n=x.size();
  int one = 1;
  double d = ddot_(n,&x[0],one,&y[0],one);
  std:: cout << d << "\n"; // should be 20
}
```

- Don’t forget to link against the BLAS library

Linking against Fortran libraries

- The C++ compiler automatically links against the C++ and C runtime libraries. Use the --verbose option with g++ to see what it does.
- The Fortran compiler automatically links against the Fortran runtime libraries. Use a --verbose option or similar with your Fortran compiler.
- Fortran libraries might need the Fortran runtime libraries but your C++ compiler does not know them. Hence we need to:
  - find the Fortran runtime libraries
  - add them to the link command of your C/C++ code
- Instructions that work on many common machines:
  - MacOS X: -framework Accelerate
  - Linux: -lgfortran

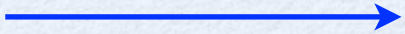
Array storage

- Fortran indices by default start at 1, while C/C++ starts at 0
- Fortran stores arrays in column-major order,
  while C/C++ uses row-major order

column-major (Fortran)

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

row-major (C/C++)



0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

- Consequence:
  - matrices are typically transposed
  - A[i][j] in C/C++ is A(j+1,i+1) in Fortran

Another look at DDOT: increments

- The DDOT dot product function takes two pointers and two increments

```
      DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
*
*     DDOT forms the dot product of two vectors.
*     uses unrolled loops for increments equal to one.
*
```

- In arrays the increments is typically 1
- The increments exist as arguments to be able to treat columns and rows in matrices as vectors

0	5	10	15	20
1	6	11	16	21
2	**7**	**12**	**17**	**22**
3	8	13	18	23
4	9	14	19	24

```
DX = start of storage + 2
INCX = 5
```

BLAS naming conventions

- BLAS functions always have one (or two) prefix indicating the type of the arguments and optional return value

I	int
S	float
D	double
C	std::complex<float>
Z	std::complex<double>

- Example: dot product

generic name	_DOT
float	SDOT
double	DDOT
std::complex<float>	CDOT
std::complex<double>	ZDOT

BLAS-1: vector operations

Reduction operations:				
s	$\leftarrow$	$x \cdot y$	inner product	__DOT_
s	$\leftarrow$	$max\{\|x_i\|\}$	pivot search	I_AMAX
s	$\leftarrow$	$\|x\|_2$	norm of a vector	_NRM2
s	$\leftarrow$	$\sum_i \|x_i\|$	sum of abs	_ASUM

Vector to vector transformations:				
y	$\leftarrow$	x	copy x into y	_COPY
x	$\leftrightarrow$	y	swap	_SWAP
y	$\leftarrow$	$\alpha \cdot x$	scale x	_SCAL
y	$\leftarrow$	$\alpha \cdot x + y$	saxpy	_AXPY

Generate and apply Givens rotations:

Compute rotation:			
$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \rightarrow \begin{bmatrix} r \\ 0 \end{bmatrix}$	$c, s \ni r = \sqrt{a^2 + b^2}$	_ROTG
Apply rotation:		
$\begin{bmatrix} x \\ y \end{bmatrix} \leftarrow \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$		_ROT

BLAS matrix types and naming conventions

- BLAS 2 and BLAS 3 support various matrix types, given as two letters after the prefix.

GE	general dense matrix
GB	banded matrix, stored packed
SY	symmetric, stored like a general dense matrix
SP	symmetric, stored packed
SB	symmetric banded, stored packed
HE	hermitian, stored like a general dense matrix
HP	hermitian, stored packed
HB	hermitian banded, stored packed
TR	upper or lower triangular, stored like a general dense matrix
TP	upper or lower triangular, stored packed
TB	upper or lower triangular band matrix, stored packed

- Example: DGEMV is matrix-vector multiplication for a general matrix of doubles

Packed storage formats

- Recall for banded matrices:

Dense storage of matrix	Packed storage as a packed matrix
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

- For triangular matrices, depending on the UPLO parameter:

UPLO	Dense storage of matrix	Packed storage as array
U	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11}\ \underbrace{a_{12}\ a_{22}}\ \underbrace{a_{13}\ a_{23}\ a_{33}}\ \underbrace{a_{14}\ a_{24}\ a_{34}\ a_{44}}$
L	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\ a_{21}\ a_{31}\ a_{41}}\ \underbrace{a_{22}\ a_{32}\ a_{42}}\ \underbrace{a_{33}\ a_{43}}\ a_{44}$

- Symmetric and hermitian packed formats store only one triangle

Dense matrix storage

- It's a bit more complicated than you thought
  - Fortran-77 and earlier did not allow dynamical allocation
  - One might want to operate just on a submatrix
- Matrix operations accept three size arguments:
  - matrix size: rows and columns of the matrix
  - leading dimension: increment between columns

0	**5**	**10**	15	20
1	**6**	**11**	16	21
2	**7**	**12**	17	22
3	8	13	18	23
4	9	14	19	24

number of rows: **3**
number of columns: **3**
leading dimension: **5**

BLAS-2: matrix-vector operations

Matrix times Vector				
x	$\leftarrow$	$\alpha Ax + \beta y$	general	_GEMV
			general band	_GBMV
			general hermitian	_HEMV
			hermitian banded	_HBMV
			hermitian packed	_HPMV
			general symmetric	_SYMV
			symmetric banded	_SBMV
			symmetric packed	_SPMV
x	$\leftarrow$	Ax	triangular	_TRMV
			triangular banded	_TBMV
			triangular packed	_TPMV

Rank one and rank two updates:				
A	$\leftarrow$	$\alpha xy^T + A$	general	_GER_
A	$\leftarrow$	$\alpha xx^* + A$	general hermitian	_HER
			hermitian packed	_HPR
A	$\leftarrow$	$\alpha(xy^* + yx^*) + A$	gen. Hermitian	_HER2
			hermitian packed	_HPR2
A	$\leftarrow$	$\alpha xx^T + A$	general symmetric	_SYR
			symmetric packed	_SPR
A	$\leftarrow$	$\alpha(xy^T + yx^T) + A$	gen. symmetric	_SYR2
			symmetric packed	_SPR2

Triangular solve:				
x	$\leftarrow$	$A^{-1}x$	triangular	_TRSV
			triangular banded	_TBSV
			triangular packed	_TPSV

BLAS-3 matrix-matrix operations

Matrix product:				
C	$\leftarrow$	$\alpha A \cdot B + \beta C$	general	_GEMM
			symmetric	_SYMM
			hermitian	_HEMM
B	$\leftarrow$	$\alpha A \cdot B$	triangular	_TRMM
Rank k update:				
C	$\leftarrow$	$\alpha A \cdot A^T + \beta C$		_SYRK
C	$\leftarrow$	$\alpha A \cdot A^H + \beta C$		_HERK
C	$\leftarrow$	$\alpha(A \cdot B^T + B \cdot A^T) + \beta C$		_SYRK2
C	$\leftarrow$	$\alpha(A \cdot B^H + B \cdot A^H) + \beta C$		_HERK2
Triangular solve for multiple r.h.s.:				
B	$\leftarrow$	$\alpha A^{-1} \cdot B$	triangular	_TRSM

Transpose arguments

- _GEMV, _GBMV, _T_MV, and _T_SV take arguments indicating whether the matrix should be transposed

TRANS	real matrix S,D	complex matrix C,Z
'N' or 'n'	no transpose	no transpose
'T' or 't'	transposed	transposed
'C' or 'c'	transposed	transposed and complex conjugated

- Similarly some of the BLAS-3 calls take one or two transpose arguments:
  - _GEMM,_TRMM
  - _SYRK, _HERK, _SY2RK,
  - _TRSM

Optimizing linear algebra operations

- BLAS-1 is best optimized by SIMD vectorization

  we will optimize at _DOT and _SCAL and look at I_AMAX

- BLAS-2 and BLAS-3 build on top of BLAS-1
  - reuse all optimizations done for BLAS-1
  - potential for further optimization by multithreading

  we will optimize _GEMV
  you will optimize _GEMM

- Other libraries, like LAPACK, are built on top of BLAS
  - reuse all optimizations done for BLAS-1, 2 and 3
  - further parallelization may be possible

  we will optimize Gaussian elimination (_GEFA)

Parallelizing _GEMV

- We have two loops in _GEMV over i and j

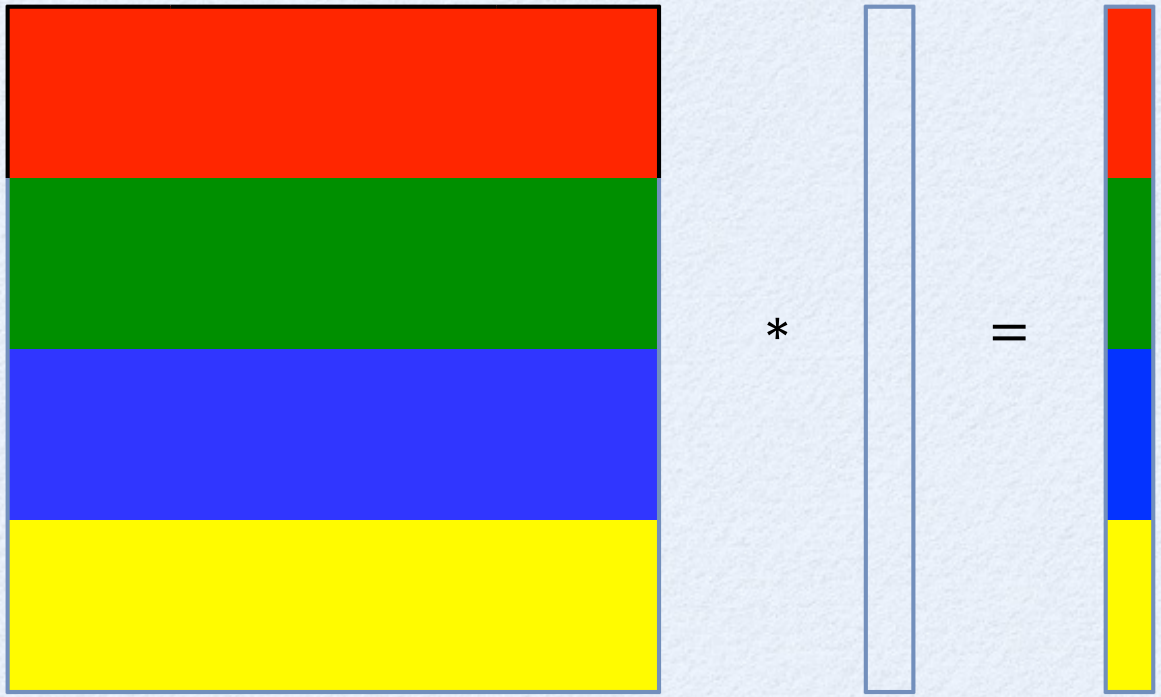
$$y_i = \sum_j A_{ij} x_j$$

- Four versions:
  - loop order can be i,j or j,i
  - either the inner or the outer loop can be parallelized
- Two more versions:
  - split the matrix into blocks and use a single-threaded BLAS _GEMV for each block
  - hope for a parallel BLAS and just call _GEMV

Case 1: i,j, parallelizing outer loop

- Parallelize the outer loop over *i*

```
#pragma omp parallel for
for (int i=0; i<M; i++) {
  y[i] = 0.;
  for (int j=0; j<N; j++)
    y[i] += A(i,j) * x[j];
}
```

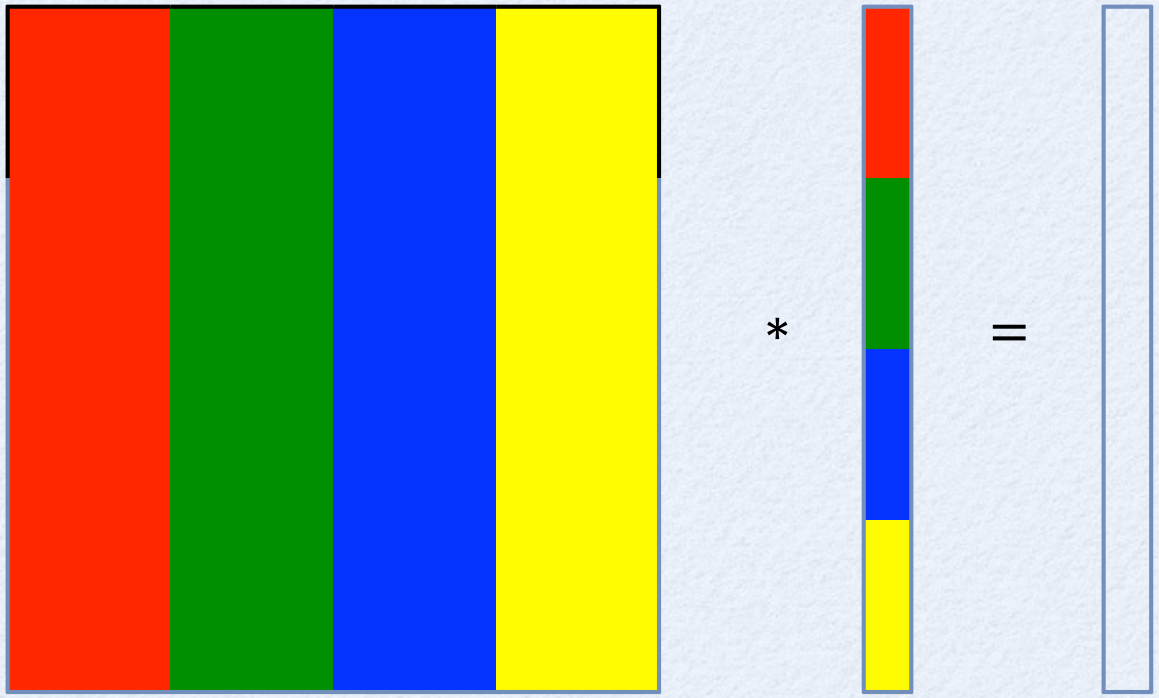


Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 2: i,j, parallelizing inner loop

- Parallelize the inner loop over *j*

```
for (int i=0; i<M; i++) {
  double tmp = 0.
  #pragma omp parallel for reduction(+ : tmp)
  for (int j=0; j<N; j++)
    tmp += A(i,j) * x[j];
  y[i] = tmp;
}
```

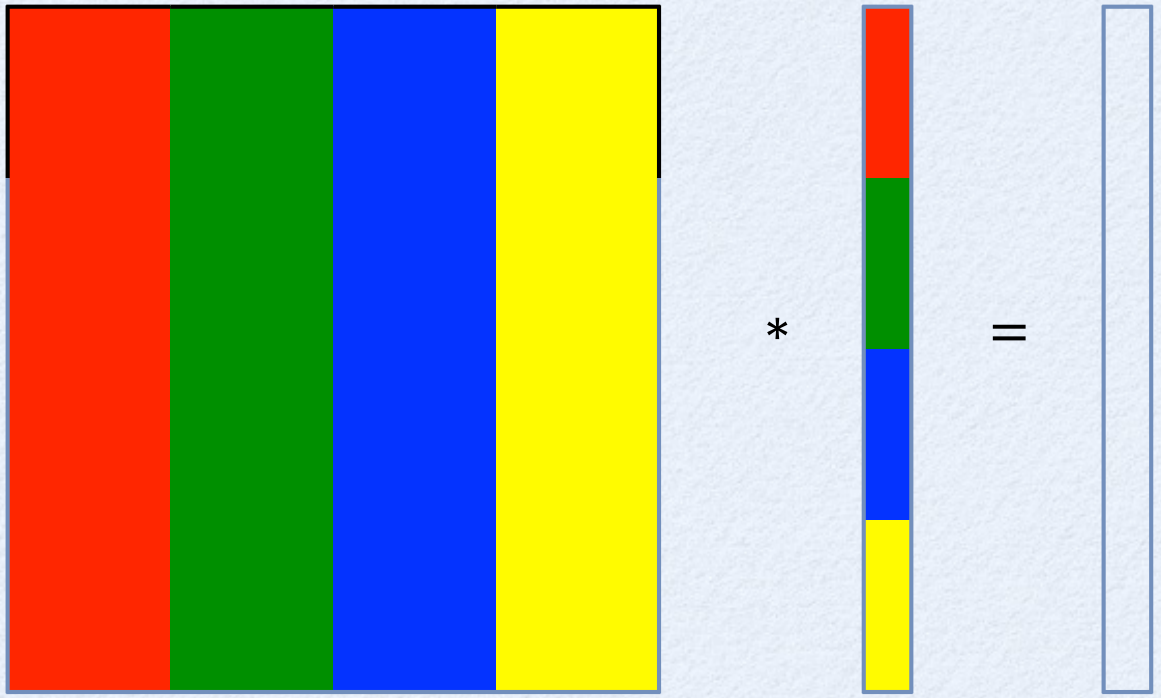


Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 3: j,i, parallelizing outer loop

- Parallelize the outer loop over *j*
- followed by a vector reduction

```
std::fill(y.begin(),y.end(),0.);
double z[M];
#pragma omp parallel private(z)
{
  std::fill(z,z+M,0.);
  #pragma omp for
  for (int j=0; j<N; j++)
    for (int i=0; i<M; i++)
      z[i] += A(i,j) * x[j];
  #pragma omp critical
  for (int i=0; i<M; i++)
    y[i] += z[i];
}
```

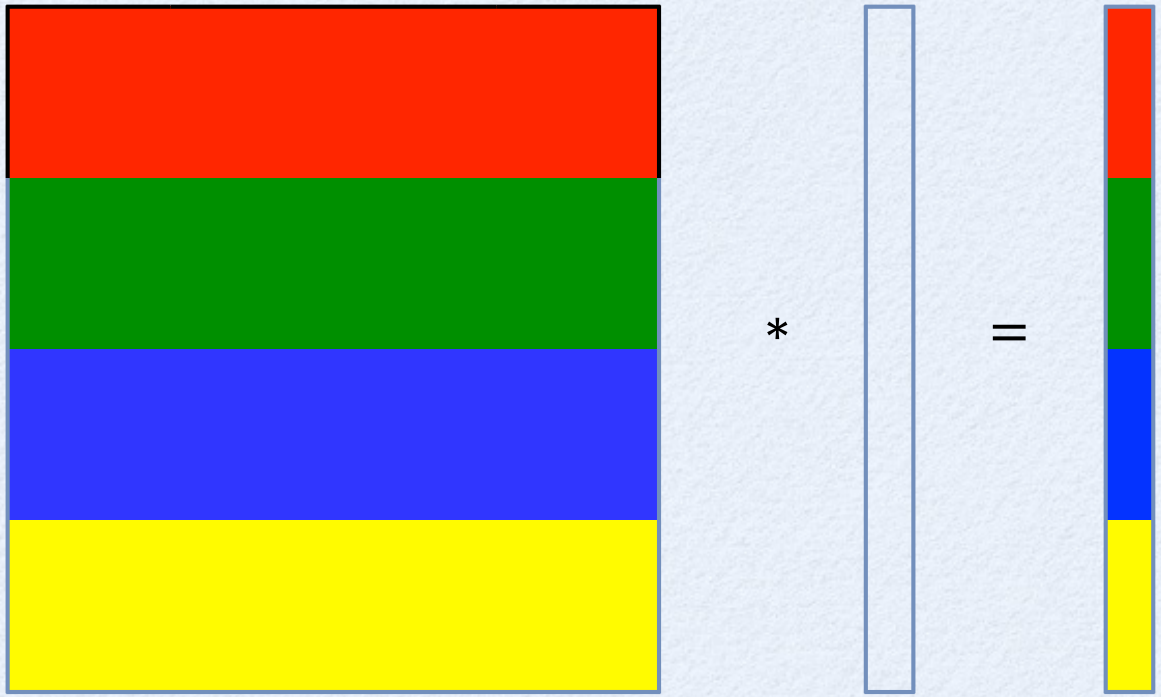


Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 4: j,i, parallelizing inner loop

- Parallelize the inner loop over *i*

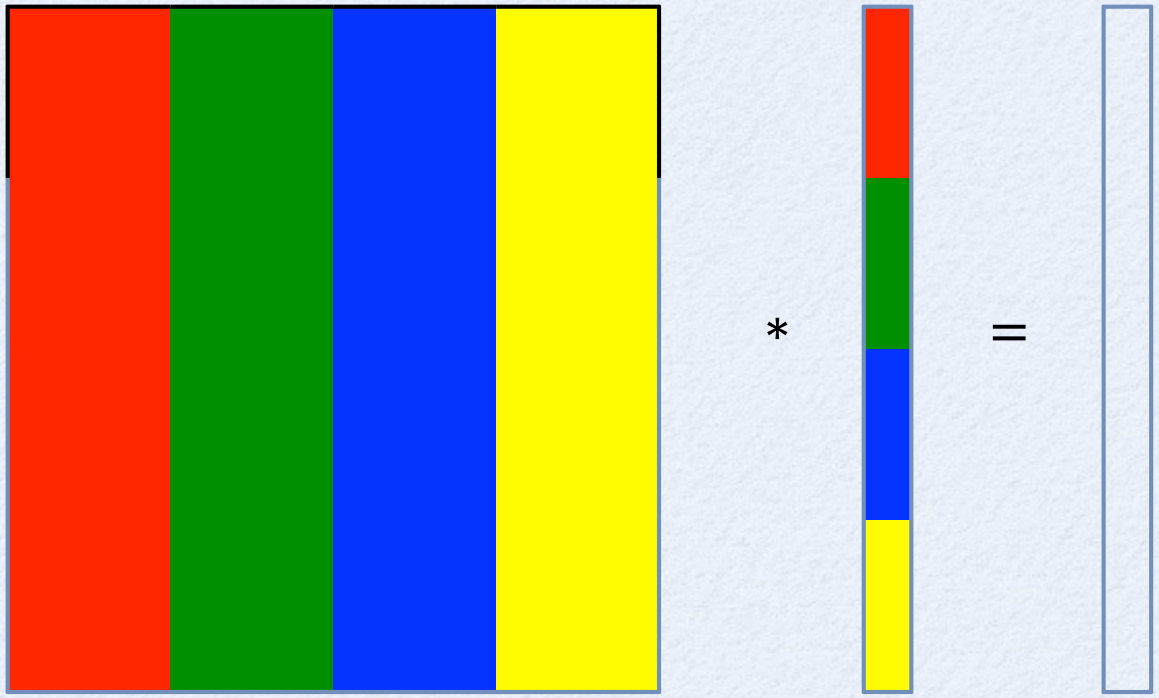
```
for (int i=0; i<M; i++)
  y[i] = 0.;

for (int j=0; j<N; j++)
  #pragma omp parallel for
  for (int i=0; i<M; i++)
    y[i] += A (i,j) * x[j];
```



Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 5: calling BLAS from every thread

```
double DONE  = 1.;
double DZERO = 0.;
int    ONE   = 1;
int    lda   = N;
#pragma omp parallel
{
  int p = omp_get_num_threads();
  int n0 = (M + p - 1)/p;
  #pragma omp for
  for (int i=0; i<p; i++) {
    int n1 = std::min(n0, M-i*n0);
    dgemv_("N", n1, n, DONE, A.data()+i*n0, lda, &x[0], ONE, DZERO, &y[i*n0], ONE);
  }
}
```



Colors indicate splitting of the matrix over threads for the case of 4 threads

What code is the fastest?

- Give us your best guess. Which code is the fastest?

	N=2000, g++ MacBook Pro	N=2000, iCC MacBook Pro		Your machine
Case 1	9.5 ms	9.4 ms		
Case 2	70 ms	67 ms		
Case 3	1.67 ms	1.57 ms		
Case 4	119 ms	5.2 ms		
Case 5	2.1 ms			
Case 6	1.58 ms			

Gaussian elimination / LU factorization

Recall how we solve dense linear systems of equations by Gaussian elimination

1. start with system of equations

$$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$$

2. swap largest coefficient of x to the first row (pivot) and store the pivot

```
pivot = [ 2 ]
```

$$\begin{aligned} -3x - y + 2z &= -11 \\ 2x + y - z &= 8 \\ -2x + y + 2z &= -3 \end{aligned}$$

3. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 \\ -2/3 \\ 2/3 \end{pmatrix}$$

$$\begin{aligned} -3x - y + 2z &= -11 \\ \left(2 - \frac{2}{3}3\right)x + \left(1 - \frac{2}{3}\right)y + (-1 + \frac{2}{3}2)z &= 8 - \frac{2}{3}11 \\ \left(-2 + \frac{2}{3}3\right)x + \left(1 + \frac{2}{3}\right)y + \left(2 - \frac{2}{3}2\right)z &= -3 + \frac{2}{3}11 \end{aligned}$$

Gaussian elimination / LU factorization

Recall how we solve dense linear systems of equations by Gaussian elimination

1. start with system of equations

$$2x + y - z = 8$$
$$-3x - y + 2z = -11$$
$$-2x + y + 2z = -3$$

2. swap largest coefficient of x to the first row (pivot) and store the pivot

```
pivot = [ 2 ]
```

$$-3x - y + 2z = -11$$
$$2x + y - z = 8$$
$$-2x + y + 2z = -3$$

3. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 \\ -2/3 \\ 2/3 \end{pmatrix}$$

$$-3x - y + 2z = -11$$
$$\frac{1}{3}y + \frac{1}{3}z = \frac{2}{3}$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$

Gaussian elimination / LU factorization

Recall how we solve dense linear systems of equations by Gaussian elimination

4. now continue with the next row

$$-3x - y + 2z = -11$$
$$\frac{1}{3}y + \frac{1}{3}z = \frac{2}{3}$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$

5. swap largest coefficient of x to the current row (pivot) and store the pivot

```
pivot = [ 2, 3 ]
```

$$-3x - y + 2z = -11$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$
$$\frac{1}{3}y + \frac{1}{3}z = \frac{2}{3}$$

6. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 & 0 \\ -2/3 & 0 \\ 2/3 & 1/5 \end{pmatrix}$$

$$-3x - y + 2z = -11$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$
$$\left(\frac{1}{3} - \frac{1}{5}\frac{5}{3}\right)y + \left(\frac{1}{3} - \frac{1}{5}\frac{2}{3}\right)z = \frac{2}{3} - \frac{1}{5}\frac{13}{3}$$

Gaussian elimination / LU factorization

Recall how we solve dense linear systems of equations by Gaussian elimination

4. now continue with the next row

$$-3x - y + 2z = -11$$
$$\frac{1}{3}y + \frac{1}{3}z = \frac{2}{3}$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$

5. swap largest coefficient of x to the current row (pivot) and store the pivot

```
pivot = [ 2, 3 ]
```

$$-3x - y + 2z = -11$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$
$$\frac{1}{3}y + \frac{1}{3}z = \frac{2}{3}$$

6. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 & 0 \\ -2/3 & 0 \\ 2/3 & 1/5 \end{pmatrix}$$

$$-3x - y + 2z = -11$$
$$\frac{5}{3}y + \frac{2}{3}z = \frac{13}{3}$$
$$\frac{1}{5}z = -\frac{1}{5}$$

7. finally solve the last equation and substitute backwards

Gaussian elimination / LU factorization

Now let us look at the coefficient matrices

1. start with system of equations

$$\begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix} \qquad \begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$$

2. swap largest coefficient of x to the first row (pivot) and store the pivot

```
pivot = [ 2 ]
```

$$\begin{pmatrix} -3 & -1 & 2 \\ 2 & 1 & -1 \\ -2 & 1 & 2 \end{pmatrix} \qquad \begin{aligned} -3x - y + 2z &= -11 \\ 2x + y - z &= 8 \\ -2x + y + 2z &= -3 \end{aligned}$$

3. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 \\ -2/3 \\ 2/3 \end{pmatrix}$$

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & 1/3 & 1/3 \\ 0 & 5/3 & 2/3 \end{pmatrix} \qquad \begin{aligned} -3x - y + 2z &= -11 \\ \frac{1}{3}y + \frac{1}{3}z &= \frac{2}{3} \\ \frac{5}{3}y + \frac{2}{3}z &= \frac{13}{3} \end{aligned}$$

Gaussian elimination / LU factorization

- **Now let us look at the coefficient matrices**

4. now continue with the next row

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & 1/3 & 1/3 \\ 0 & 5/3 & 2/3 \end{pmatrix}$$

$$\begin{aligned} -3x - y + 2z &= -11 \\ \frac{1}{3}y + \frac{1}{3}z &= \frac{2}{3} \\ \frac{5}{3}y + \frac{2}{3}z &= \frac{13}{3} \end{aligned}$$

5. swap largest coefficient of x to the current row (pivot) and store the pivot

```
pivot = [ 2, 3 ]
```

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & 5/3 & 2/3 \\ 0 & 1/3 & 1/3 \end{pmatrix}$$

$$\begin{aligned} -3x - y + 2z &= -11 \\ \frac{5}{3}y + \frac{2}{3}z &= \frac{13}{3} \\ \frac{1}{3}y + \frac{1}{3}z &= \frac{2}{3} \end{aligned}$$

6. divide the coefficient of *x* in other rows by that of the first row. For each row multiply the first row by that factor and subtract it from the row. Store the coefficients.

$$\begin{pmatrix} 0 & 0 \\ -2/3 & 0 \\ 2/3 & 1/5 \end{pmatrix}$$

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & 5/3 & 2/3 \\ 0 & 0 & 1/5 \end{pmatrix}$$

$$\begin{aligned} -3x - y + 2z &= -11 \\ \frac{5}{3}y + \frac{2}{3}z &= \frac{13}{3} \\ \frac{1}{5}z &= -\frac{1}{5} \end{aligned}$$

Gaussian elimination / LU factorization

- Now let us look at the matrices

Pivot

```
pivot = [ 2, 3 ]
```

Coefficients

$$\begin{pmatrix} 0 & 0 \\ -\frac{2}{3} & 0 \\ \frac{2}{3} & \frac{1}{5} \end{pmatrix}$$

Triangular

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & \frac{5}{3} & \frac{2}{3} \\ 0 & 0 & \frac{1}{5} \end{pmatrix}$$

Gaussian elimination / LU factorization

- Now let us look at the matrices
  - Expand the coefficient matrices to a lower triangular matrix *L*

Pivot

```
pivot = [ 2, 3 ]
```

Coefficients

$$L = \begin{pmatrix} 0 & 0 & 0 \\ -\frac{2}{3} & 0 & 0 \\ \frac{2}{3} & \frac{1}{5} & 0 \end{pmatrix}$$

Triangular

$$\begin{pmatrix} -3 & -1 & 2 \\ 0 & \frac{5}{3} & \frac{2}{3} \\ 0 & 0 & \frac{1}{5} \end{pmatrix}$$

Gaussian elimination / LU factorization

- Now let us look at the matrices
  - Expand the coefficient matrices to a lower triangular matrix *L*
  - Put ones on the diagonal of *L* and correct for the permutations

Pivot

```
pivot = [ 2, 3 ]
```

Coefficients

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ -\frac{2}{3} & \frac{1}{5} & 1 \end{pmatrix}$$

Triangular

$$U = \begin{pmatrix} -3 & -1 & 2 \\ 0 & \frac{5}{3} & \frac{2}{3} \\ 0 & 0 & \frac{1}{5} \end{pmatrix}$$

Gaussian elimination / LU factorization

- Now let us look at the matrices
  - Expand the coefficient matrices to a lower triangular matrix *L*
  - Put ones on the diagonal of *L*
  - Convert the pivot array into permutation matrices *L*
  - And we get an LU factorization

Pivot

```
pivot = [ 2, 3 ]
```

$$P = P_{12}P_{23} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Coefficients

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ -2/3 & 1/5 & 1 \end{pmatrix}$$

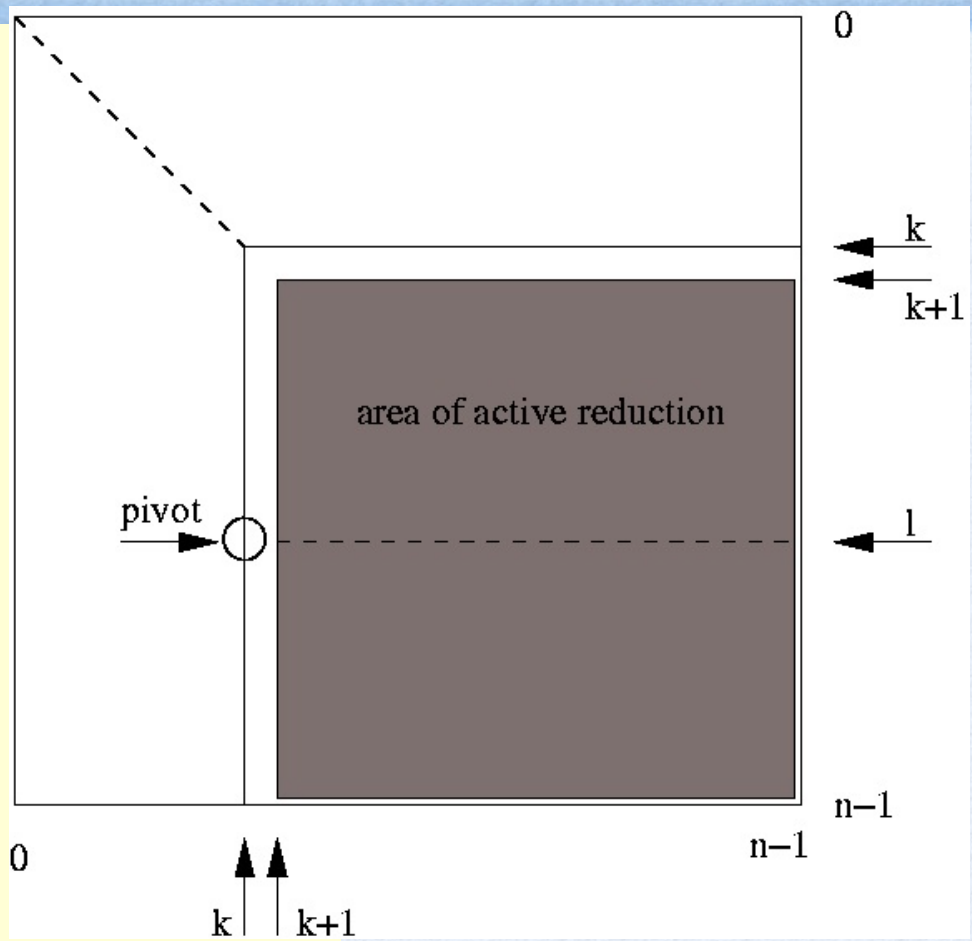
Triangular

$$U = \begin{pmatrix} -3 & -1 & 2 \\ 0 & 5/3 & 2/3 \\ 0 & 0 & 1/5 \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix} = PLU = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ -2/3 & 1/5 & 1 \end{pmatrix}\begin{pmatrix} -3 & -1 & 2 \\ 0 & 5/3 & 2/3 \\ 0 & 0 & 1/5 \end{pmatrix}$$

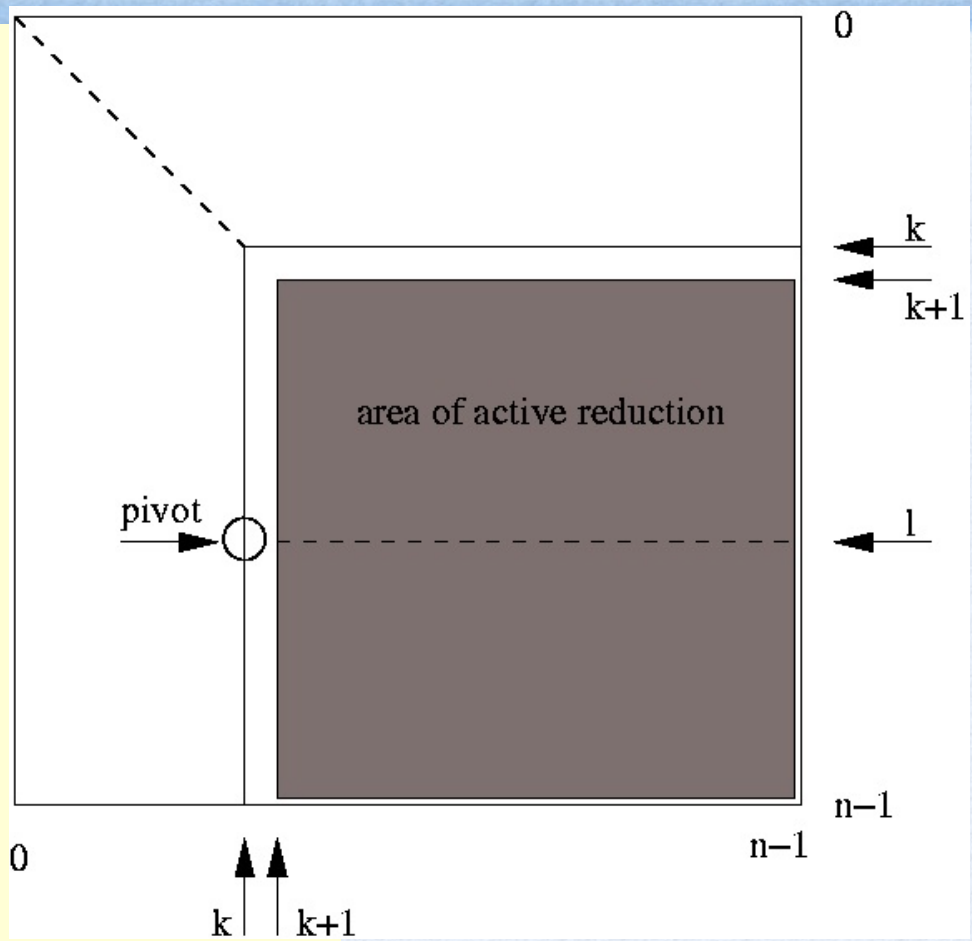
Coding a linpack-style LU factorization

```
void dgefa(hpc12::matrix<double,hpc12::column_major>& a,
           std::vector<int> pivot)
{
  assert(a.num_rows() == a.num_cols());

  pivot.clear();
  int one=1;
  int n=a.num_rows();
  int lda=a.leading_dimension();

  for(int k=0; k < a.num_rows()-1; k++){
    // 1. find the index of the largest element in column k
    //    starting at row k
    int nk = n-k;
    int l = idamax_(nk,&a(k,k),one) + k;
    pivot.push_back(l); // and save it
    assert( a(l,k) =!0.0); // error if pivot is zero

    // 2. swap rows l and k, starting at column k
    dswap_(nk,&a(l,k),lda,&a(k,k),lda);

    // 3. scale the column k below row k by the inverse
    // negative pivot element, to store L in the lower part
    double t = -1./a(k,k);
    int nmk1 = n-k-1;
    dscal_(nmk1,t,&a(k+1,k),one);

    // 4. add the scaled k-th row to all rows in the lower right corner
    for(int j=k+1; j<n; j++) {   // multiple daxpy
      double t   = a(j,k);
      daxpy_(nmk1,t,&a(k,k+1),one,&a(j,k+1),one);
    }
  }
}
```



Coding a linpack-style LU factorization

```
void dgefa(hpc12::matrix<double,hpc12::column_major>& a,
           std::vector<int> pivot)
{
  assert(a.num_rows() == a.num_cols());

  pivot.clear();
  int one=1;
  int n=a.num_rows();
  int lda=a.leading_dimension();

  for(int k=0; k < a.num_rows()-1; k++){
    // 1. find the index of the largest element in column k
    //    starting at row k
    int nk = n-k;
    int l = idamax_(nk,&a(k,k),one) + k;
    pivot.push_back(l); // and save it
    assert( a(l,k) =!0.0); // error if pivot is zero

    // 2. swap rows l and k, starting at column k
    dswap_(nk,&a(l,k),lda,&a(k,k),lda);

    // 3. scale the column k below row k by the inverse
    // negative pivot element, to store L in the lower part
    double t = -1./a(k,k);
    int nmk1 = n-k-1;
    dscal(nmk1,t,&a(k+1,k),one);

    // 4. add the scaled k-th row to all rows in the lower right corner
    #pragma omp parallel for
    for(int j=k+1; j<n; j++) {   // multiple daxpy
      double t   = a(j,k);
      daxpy(nmk1,t,&a(k,k+1),one,&a(j,k+1),one);
    }
  }
}
```

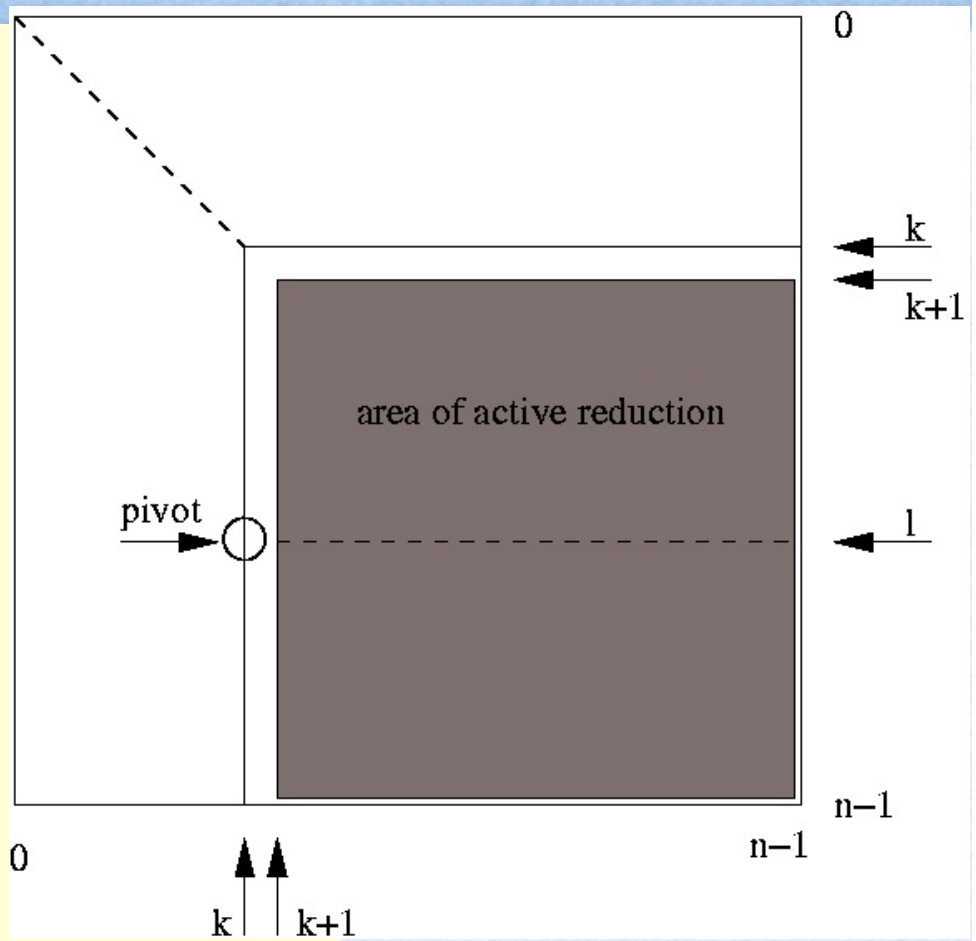


Optimizing the LU factorization

- LU factorization is the key operation in the LINPACK benchmark used to measure supercomputer performance
- All low-level vector operations are dispatched to optimized BLAS
  - DSCAL
  - DSWAP
  - IDAMAX
  - DAXPY
- The loop over many DAXPY operations can be multithreaded, e.g. by OpenMP
- Better yet: call a multi-threaded BLAS 2 outer product **DGER** if you have one, or parallelize DGER in your BLAS2, instead of putting the multithreading here

Coding a linpack-style LU factorization

```
void dgefa(hpc12::matrix<double,hpc12::column_major>& a,
           std::vector<int> pivot)
{
  assert(a.num_rows() == a.num_cols());

  pivot.clear();
  int one=1;
  int n=a.num_rows();
  int lda=a.leading_dimension();

  for(int k=0; k < a.num_rows()-1; k++){
    // 1. find the index of the largest element in column k
    //    starting at row k
    int nk = n-k;
    int l = idamax(nk,&a(k,k),one) + k;
    pivot.push_back(l); // and save it
    assert( a(l,k) =!0.0); // error if pivot is zero

    // 2. swap rows l and k, starting at column k
    dswap(nk,&a(l,k),lda,&a(k,k),lda);

    // 3. scale the column k below row k by the inverse
    // negative pivot element, to store L in the lower part
    double t = -1./a(k,k);
    int nkm1 = n-k-1;
    dscal(nkm1,t,&a(k+1,k),one);

    // 4. add the scaled k-th row to all rows in the lower right corner
    double alpha=1.;
    dger_(nmk1,nmk1,alpha,&a(k+1,k),one,&a(k,k+1),lda,&a(k+1,k+1),lda);
  }
}
```



What do we need to optimize?

- Write the fastest possible BLAS-1 and BLAS-2 functions
- Are we optimal now?
  - No, BLAS-2 performs N^2 operations on N^2 data
  - We are limited by memory-band width
- What can we do?
  - Rewrite DGEFA to use BLAS-3
  - If we can use matrix multiplication DGEMM instead of vector outer product DGER then we can reuse data, perform $O(N^3)$ operations on $O(N^2)$ data and get peak performance.
  - Do you have ideas how we can do that?