Overview of Parallel Maths Libraries

Gavin Pringle
gavin@epcc.ed.ac.uk

Overview

- Why Libraries?
- Linear Algebra
  - Matrix types
  - Serial Libraries
    - BLAS and LAPACK
  - Parallel Libraries
    - ScaLAPACK
- PDEs
  - PETSc
- FFTs
  - FFTW
    - Parallel FFTs

Libraries

Why Use Libraries?

- Easier
  - save effort and time, no duplicated code
- Avoid bugs
  - libraries are (hopefully) well tested
- Performance
  - usually contain efficient implementations,
  - perhaps optimised for a given system (vendor-specific architectures), or
  - auto-tuning

What is a Library?

- Collection of pre-implemented routines available in some sort of package
  - static archive, shared library, Framework, DLL, etc.
- Defined (and hopefully documented) API
- Often provided with operating system (e.g. Scientific Linux), available via package managers, or binary/source downloads
- Can be combined with your code by linking (and perhaps #including header files)

Basic Linear Algebra Matrices

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): $a_{ij}^* = a_{ji} \longrightarrow A^{T*} = A^{\uparrow} = A$

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$$

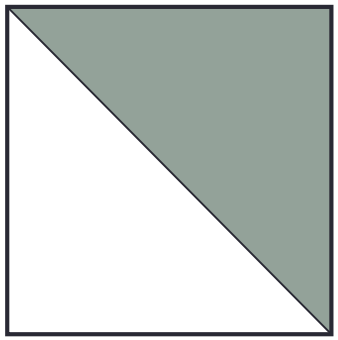
Symmetric

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

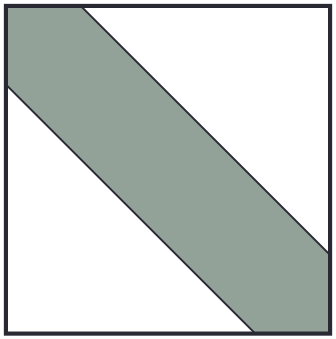
Hermitian

Matrix structures

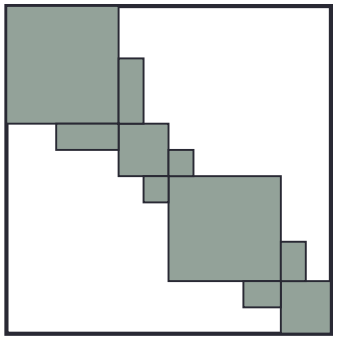
- Many matrices have a regular structure



Upper triangular



Band diagonal



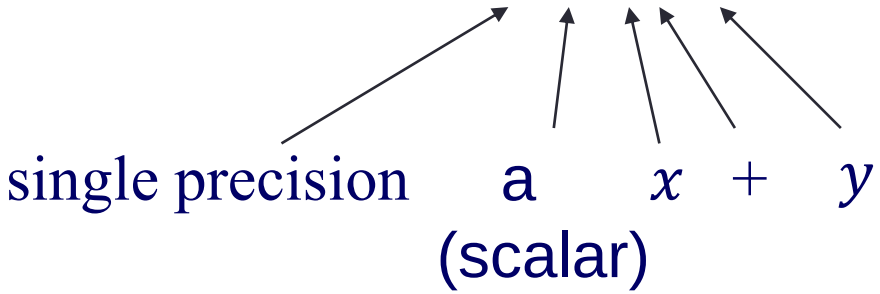
Block diagonal

Serial Linear Algebra Libraries

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



y is replaced "in-place" with $a\ x\ +\ y$

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
```

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

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 $A\,x = b$)

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. HSL (formerly the **Harwell Sparse Matrix Library**)
- In general, iterative methods are used

Parallel Dense Linear Algebra Library

Parallel dense linear algebra libraries

- ScaLAPACK is basically a parallel version of LAPACK with a few extra routines
  - E.g. matrix transposition
  - Some LAPACK routines have been improved to help with scalability

- Large **sparse** matrices: instead use **ARPACK** or parallel equivalent (**P_ARPACK**)

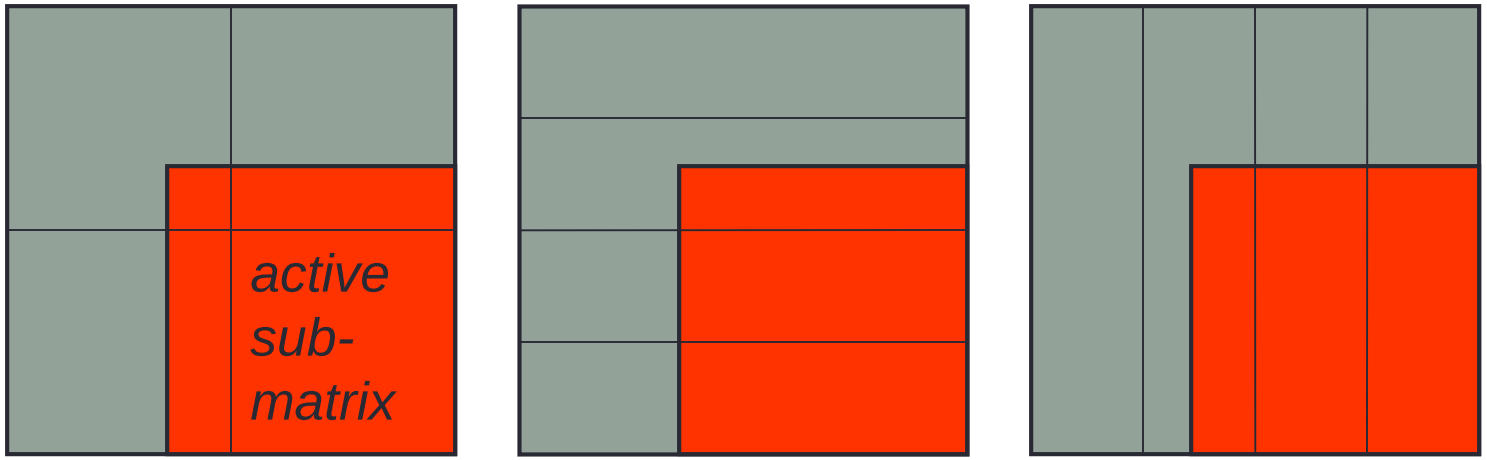
Dense Linear Algebra in parallel

- Consider LU factorisation
  - equivalently Gaussian Elimination
- Main operations were
  - `saxpy` – for subtracting one row from another
  - `dgemm` – matrix-matrix multiplication – scaling
- Algorithm works on progressively smaller sub-matrix
  - More zeros on each row as you move down the matrix

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & a'_{32} & a'_{33} & a'_{34} \\ 0 & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \dots$$

Parallelisation

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

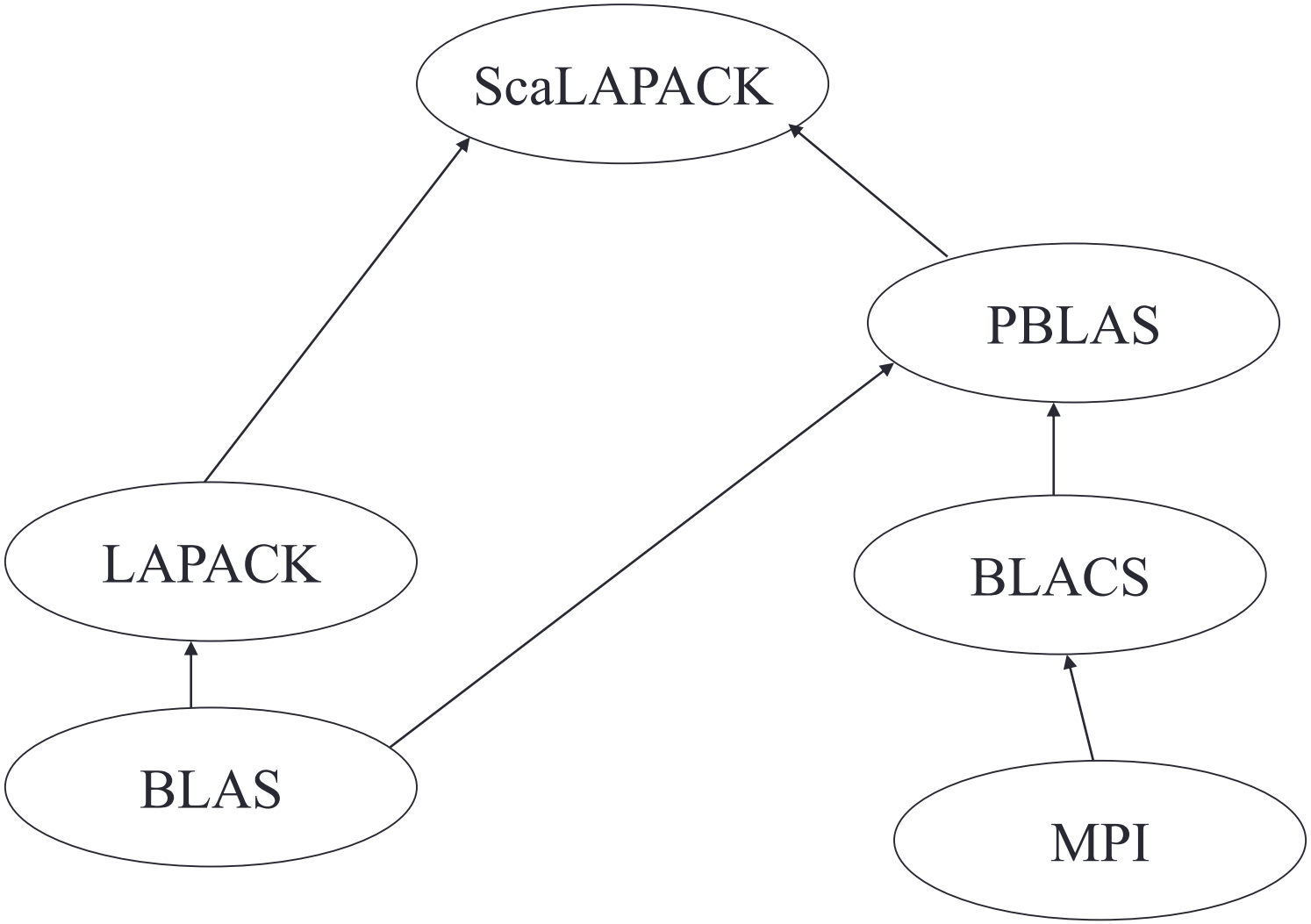


- No simple block distribution is appropriate
  - But clear we must use block-cyclic in at least one dimension

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

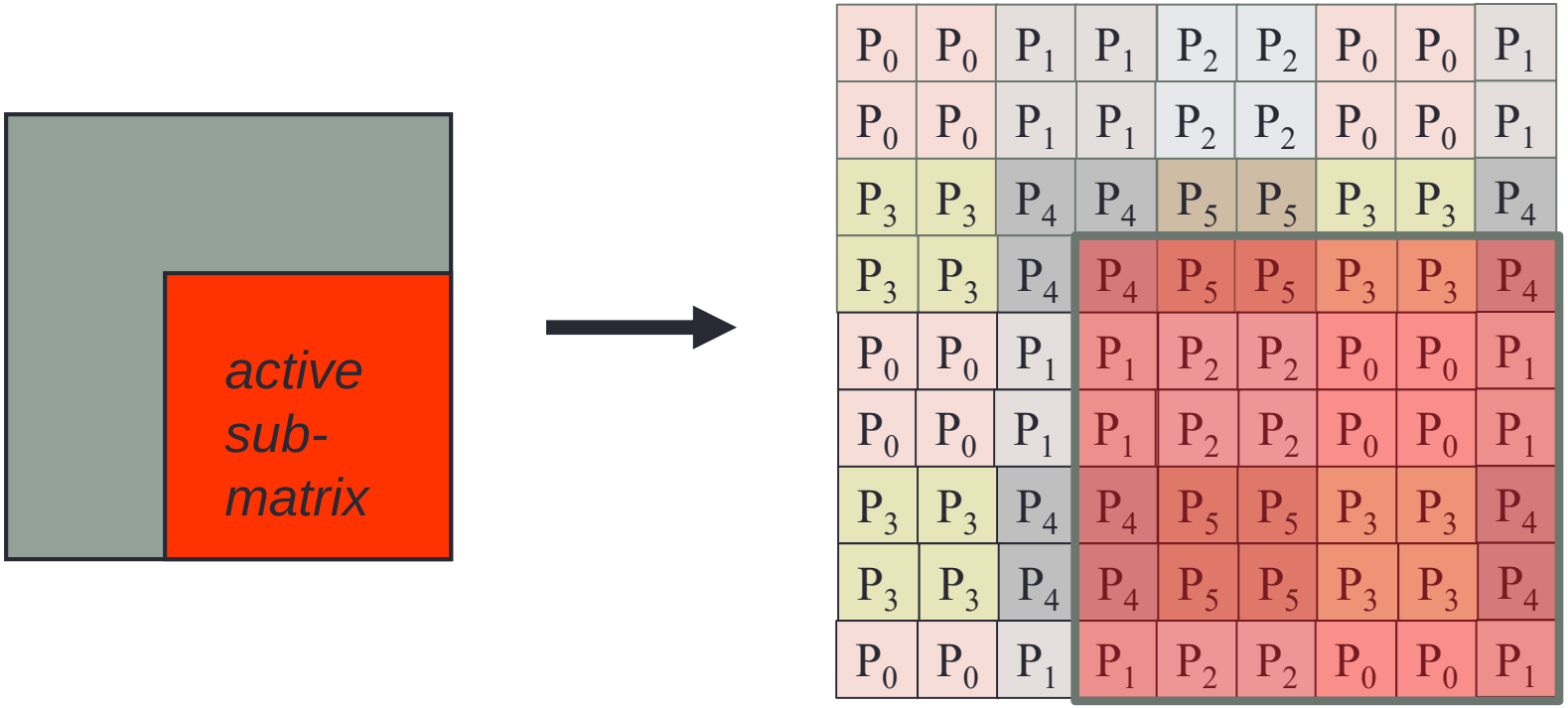
- …or by making processor blocks smaller can be more complicated…

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: 9×9
- Block size: 2×2
- No. processors: 6
- Processor grid: 2×3
- 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

2D block cyclic applied to LU factorisation



- This decomposition allows a reasonable load balance

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)
```

Brief Diversion
Calling Fortran Libraries from C

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

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
```

Parallel Partial Differential Equation solvers

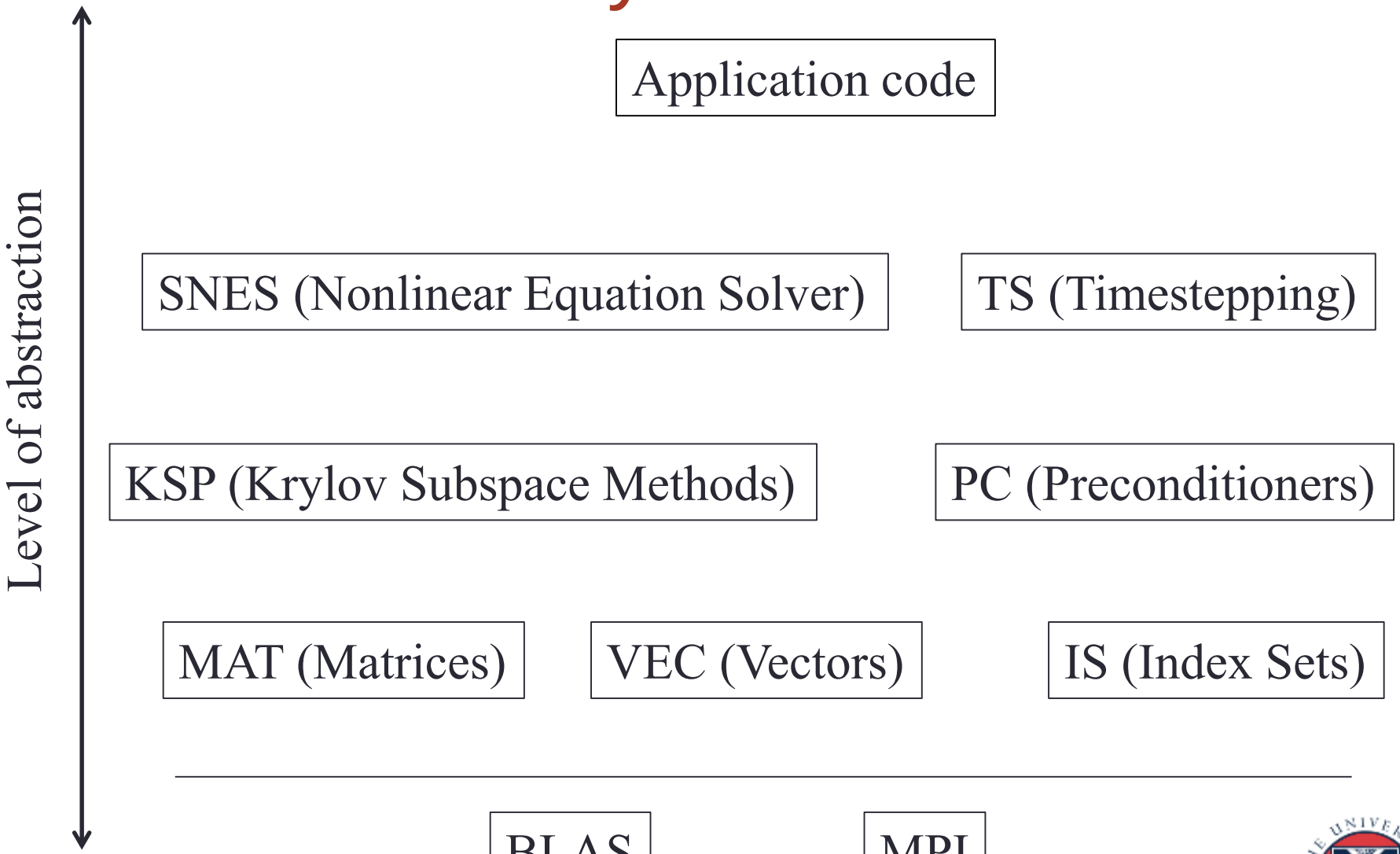
PETSc

- The Portable, Extensible Toolkit for Scientific computing
  - http://www.mcs.anl.gov/petsc/petsc-as/
  - Pronounced PET-see (the S is silent)
- Suite of data structures and routines for the parallel solution of Partial Differential Equations.
  - PDEs
- It supports MPI, and GPUs through
  - CUDA or OpenCL, as well as
  - hybrid MPI-GPU parallelism.
- Current PETSc version is 3.12

PETSc use in codes

- There are many, but notable examples
  - SLEPc
    - Scalable Library for Eigenvalue Problems
  - Fluidity
    - a finite element/volume fluids code
  - libMesh
    - adaptive finite element library
  - FEniCS and Firedrake
    - sophisticated Python based finite element simulation package
  - …

PETSc Hierarchy



Design characteristics

- Distributed memory programming only
  - No shared/global variables
- Object-oriented
- Callable from C, C++, Fortran, Python
- Same code runs in serial or parallel
- Either real or complex scalar datatype
  - library build option,
  - your code doesn't look different

Example

```c
#include <petsc.h>

int main(int argc, char **argv)
{
    Vec x;
    PetscInt n = 100;
    PetscScalar y;
    PetscInitialize(&argc, &argv,
                    PETSC_NULL,
                    PETSC_NULL);
    VecCreate(PETSC_COMM_WORLD, &x);
    VecSetSizes(x, PETSC_DECIDE, n);
    VecSetFromOptions(x);
    VecSetRandom(x, PETSC_NULL);
    VecDot(x, x, &y);

    PetscPrintf(PETSC_COMM_WORLD,
                "Dot product is %g\n", y);
    VecDestroy(&x);
    PetscFinalize();
    return 0;
}
```

```fortran
program main
#include "finclude/petscdef.h"
  use petsc
  implicit none
  Vec :: x
  PetscInt :: n = 100
  PetscScalar :: y
  integer :: ierr
  character(len=12) :: tmp
  call PetscInitialize(PETSC_NULL_CHARACTER, &
       ierr)
  call VecCreate(PETSC_COMM_WORLD, x, ierr)
  call VecSetSizes(x, PETSC_DECIDE, n, ierr)
  call VecSetFromOptions(x, ierr)
  call VecSetRandom(x, PETSC_NULL_OBJECT, ierr)
  call VecDot(x, x, y, ierr)

  write(tmp, '(F10.5)')y
  call PetscPrintf(PETSC_COMM_WORLD, &
       'Dot product is'//trim(tmp)//'\n', &
       ierr)
  call VecDestroy(x, ierr)
  call PetscFinalize(ierr)
end program main
```

epcc

THE UNIVERSITY OF EDINBURGH

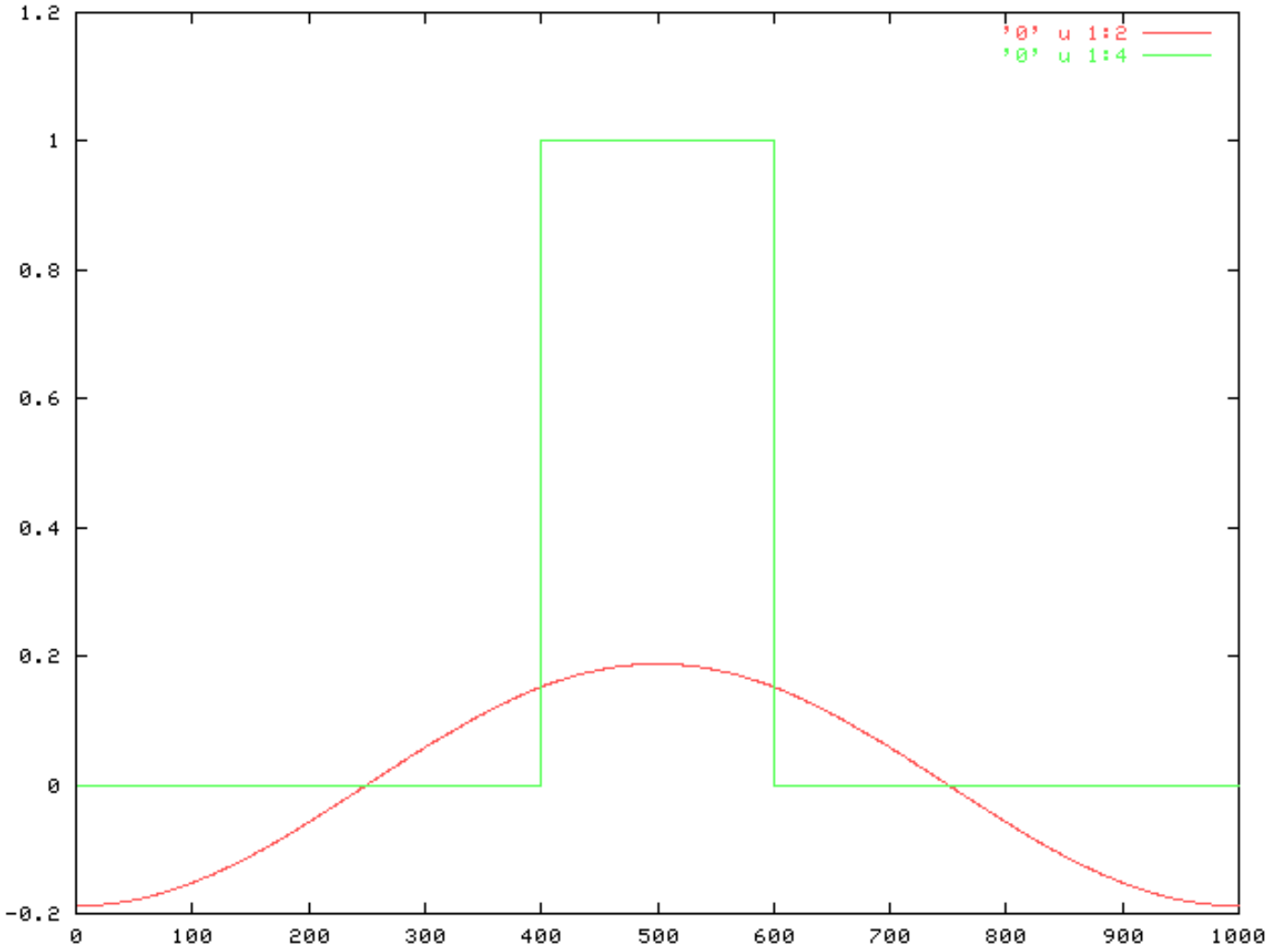
Parallel Fast Fourier Transforms

Who would use Fourier Transforms?

- Physical Sciences
  - Cosmology (P^3M N-body solvers)
  - Fluid mechanics
  - Computational Chemistry
  - Quantum physics
  - Signal and image processing
    - Antenna studies
    - Optics
- Numerical analysis
  - Linear systems analysis
  - Boundary value problems
  - Large integer multiplication (Prime finding)
- Statistics
  - Random process modelling
  - Probability theory

Caveat: different disciplines use different notation, normalisation, and sign conventions

Example: The Top Hat Function



Mathematics of the Fourier Transform

- The Fourier Transform of a complex function $f(x)$

$$F(s) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi xs} dx$$

- The inverse Fourier Transform

$$f(x) = \int_{-\infty}^{\infty} F(s) e^{i2\pi xs} ds$$

Discrete Fourier Transform

- The Discrete Fourier Transform of N complex points f_k is defined as

$$F_n = \sum_{k=0}^{N-1} f_k e^{2\pi i k n / N}$$

- The inverse Discrete Fourier Transform, which recovers the set of f_k values exactly from the F_n values is

$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{-2\pi i k n / N}$$

Discrete Fourier Transform

- The DFT can be rewritten as

$$F_n = a_0 + \sum_{k=1}^{N-1} \left(a_k \cos\left(2\pi k \frac{n}{N}\right) + b_k i \sin\left(2\pi k \frac{n}{N}\right) \right)$$

- Thus, the DFT essentially returns real number values for a_k and b_k, stored in a complex array
  - a_k and b_k are functions of f_k
  - remaining trigonometric constants (twiddle factors) may be pre-computed for a given N

Fast Fourier Transform

- In 1965, J.W. Cooley and J.W. Tukey published a DFT algorithm which is of $\mathcal{O}(N \log N)$
  - Fast Fourier Transform (FFT)
- N must be a power of 2
  - FFTs in general are not limited to powers of 2, however, the order may resort to $\mathcal{O}(N^2)$
  - Essentially a divide-and-conquer algorithm
  - In hindsight, faster than $\mathcal{O}(N^2)$ algorithms were previously, independently discovered
    - Gauss was probably first to use such an algorithm in 1805

FFT Libraries

- FFTs do not normalise
  - Each FFT/Inverse FFT pair scales by a factor of N
  - Usually left as an exercise for the programmer.
- DFTs are complex-to-complex transforms, however, most applications require real-to-complex transforms
  - Simple solution: set imaginary part of input data to be zero
    - This will be relatively slow
  - May be better to pack and unpack data
    - Place all the real data into all slots of the input, complex array (of length ($N/2$) and then unpack the result on the other side ($\mathcal{O}(N)$)
    - Around twice as fast as the simple solution
    - Good details in Numerical Recipes book
  - Some libraries have real-to-complex wrappers
  - Similar approach allows us to do 2 independent real-to-complex transforms simultaneously

FFT Libraries

- Multidimensional FFTs
  - simply successive FFTs over each dimension
    - order immaterial: linearly independent operations
  - can place data into 1D array
  - strided FFTs
  - some libraries have multidimensional FFT wrappers
- Parallel FFTs
  - performing FFT on distributed data
  - 1D FFTs are cumbersome to parallelise
    - Suitable only for huge N
  - 2D, 3D, 4D, …, parallel, array transpose operation
    - distributed data is collated on one processor before FFT
    - more in parallel libraries lecture later
  - some libraries have parallel FFT wrappers

FFTW

- Fastest Fourier Transform in the West
  - www.fftw.org
- The FFTW package was developed at MIT by Matteo Frigo and Steven G. Johnson
- Free under GNU General Public License
- Portable, self-optimising C code
  - Runs on a wide range of platforms
- Arbitrary sized FFTs of one or more dimensions
  - Fastest routines where N is composed of
    - powers of 2, 3, 5 and 7 and
    - multiples of 11 and 13, and
    - other sizes can be optimised for at configuration time

FFTW: FFTW2 vs FFTW3

- Previous version: “FFTW2”
  - Many legacy codes employ FFTW2
  - Simple(r) C interface, with wrappers for many other languages
  - Supports MPI
  - Rest of this lecture assumes “FFTW2”

- New version: “FFTW3”
  - Different interface to FFTW2 – to allow planner more freedom to optimise
    - Users must rewrite code
  - Supports MPI (now)
  - Somewhat faster than FFTW2 (~10% or more)

FFTW: Details

- Can perform FFTs on distributed data
  - MPI for distributed memory platforms
  - OpenMP or POSIX for SMPs
- If users rewrite their code to this FFT just once then the user is saved from
  - learning platform dependent, proprietary FFT routines
  - rewriting their code every time they port their code
    - No standard interface to FFTs
  - drastically rewriting their makefiles
    - location of FFTW libraries may vary
- FORTRAN wrappers for the majority of routines
- FFTs can not always be done “in-place”
    - The input and output arrays must be separate and distinct or temp arrays created

FFTW: Plans

- All FFT libraries pre-compute the *twiddle factors*
- FFTW 'plans' also generates the FFT code from *codelets*
  - Codelets compiled when FFTW configured
- Two forms of plans
  - Estimated
    - The best numerical routines are guessed, based on information gleaned from the configuration process.
  - Measured
    - Different numerical routines are actually run and timed with the fastest being used for all future FFTW calls using this plan.
- Old plans can be reused or even read from file: *wisdom*

FFTW: Installation

- Download library from the website and unpack
  - gzipped tar file
- **`./configure; make; make install`**
  - Probes the local environment
  - Compiles many small C object codes called *codelets*
  - User can provide non-standard compiler optimisation flags
  - Libraries (both static and dynamic) are then installed along with online documentation and header files
- Includes test suite
  - Very important for any numerical library
- Pre-installed with Scientific Linux

FFTW: Compiling code

- **`gfortran fft_code.f -O3 -lfftw`**
  - If using C, FFTW must be linked with **`-lfftw -lm`**
  - If the FFTW library is configured for both single and double precision, then link with **`-lsfftw`** and **`-ldfftw`**, respectively.
- Example FORTRAN code:

```
integer :: plan
integer, parameter :: n = 1024
complex :: in(n), out(n)
! plan the computation
call fftw_f77_create_plan(…)
! execute the plan
call fftw_f77_one(…)
```

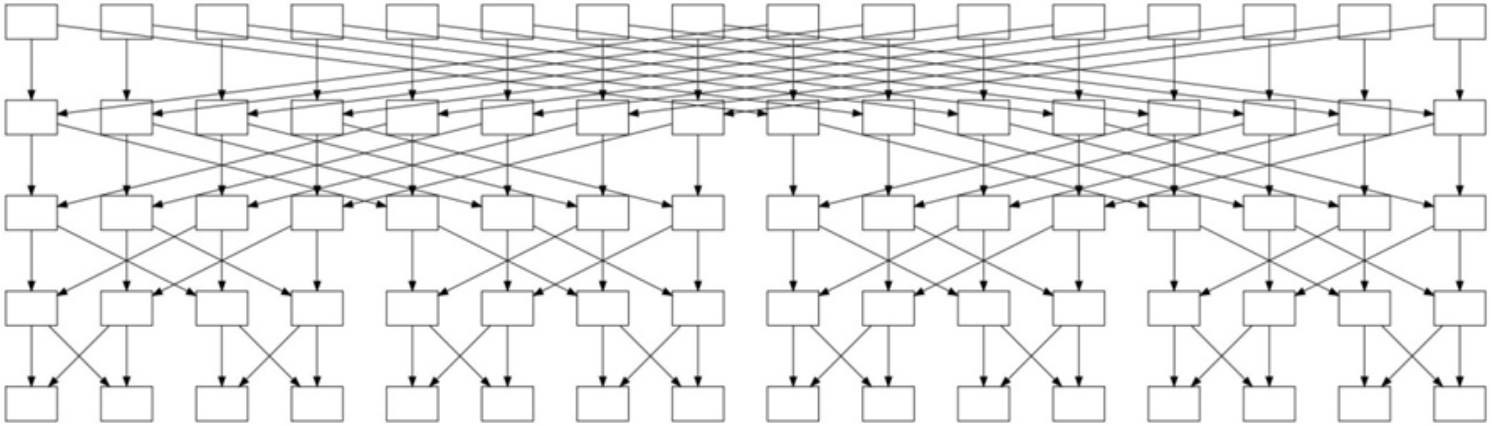
  - NB: actual incantations are not given here as reading documentation is integral to utilising any numerical library

FFTW: Performance

- The FFTW homepage, www.fftw.org, details the performance of the library compared to proprietary FFTs on a wide range of platforms.
- The FFTW library is faster than any other portable FFT library
- Comparable with machine-specific libraries provided by vendors
- Performance results from http://www.fftw.org/speed/

Parallel 1D FFT (1/2)

- Parallelisation of a 1D FFT is hard
  - Combining of data requires a lot of inter-processor communication



- Typically **N≈100–200** in many scientific codes e.g. materials chemistry – small amount of data
- Algorithm is hard to decompose
- Literature examples:

Franchetti, Voronenko, Püschel, "FFT Program Generation for Shared Memory: SMP and Multicore", Paper presented at SC06, Tampa, FL
http://sc06.supercomputing.org/schedule/pdf/pap169.pdf

Tang et al, "A Framework for Low-Communication 1-D FFT", SC12,
https://software.intel.com/sites/default/files/bd/8b/fft-1d-framework.pdf

Parallel 1D parallel FFT (2/2)

- Parallelisation works for large problems only ☹
- Sensitive to contention (shared buses) ☹
- Multicore chips with communications at cache level appear beneficial – might "be there" in a few years time
- Shows speedup, but not always "perfect" 😐
- Presently: **1D FFT is an "expensive sum" of an array which is hard to parallelise**

FFTs in two dimensions

- What needs calculating for a 2D FFT:

$$\tilde{f}(k,l) = \sum_{y=1}^{M} \left\{ \sum_{x=1}^{N} \left[f(x,y) \exp\left(-2\pi \mathrm{i} \frac{kx}{N}\right) \right] \exp\left(-2\pi \mathrm{i} \frac{ly}{M}\right) \right\}$$

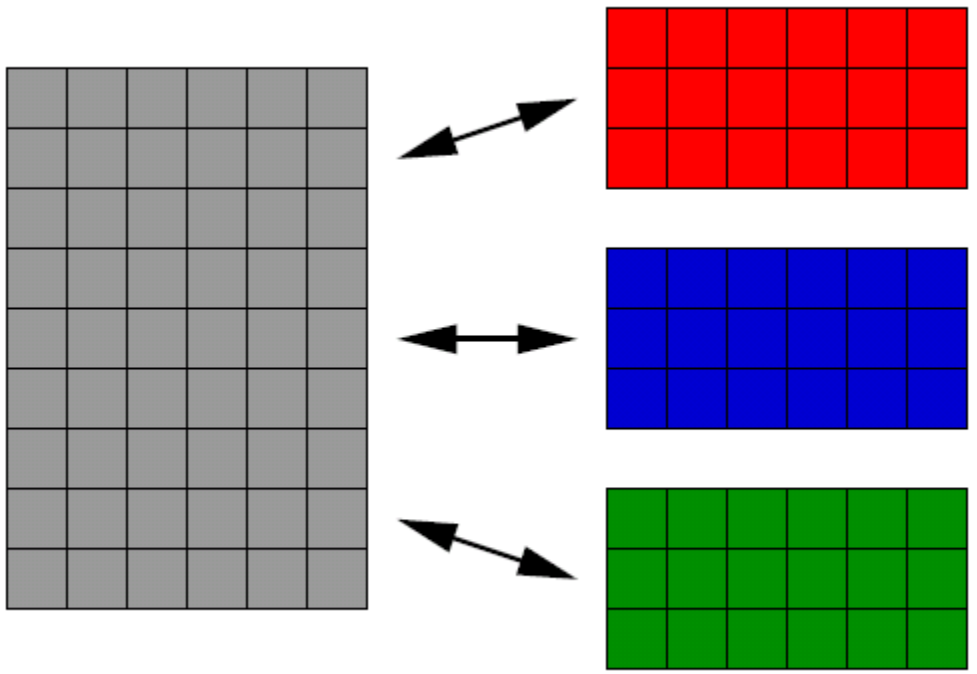
- Do it in a 2-step approach:

$$\hat{f}(k,y) \equiv \sum_{x=1}^{N} \left[f(x,y) \exp\left(-2\pi \mathrm{i} \frac{kx}{N}\right) \right]$$

$$\tilde{f}(k,l) = \sum_{y=1}^{M} \left\{ \hat{f}(k,y) \exp\left(-2\pi \mathrm{i} \frac{ly}{M}\right) \right\}$$

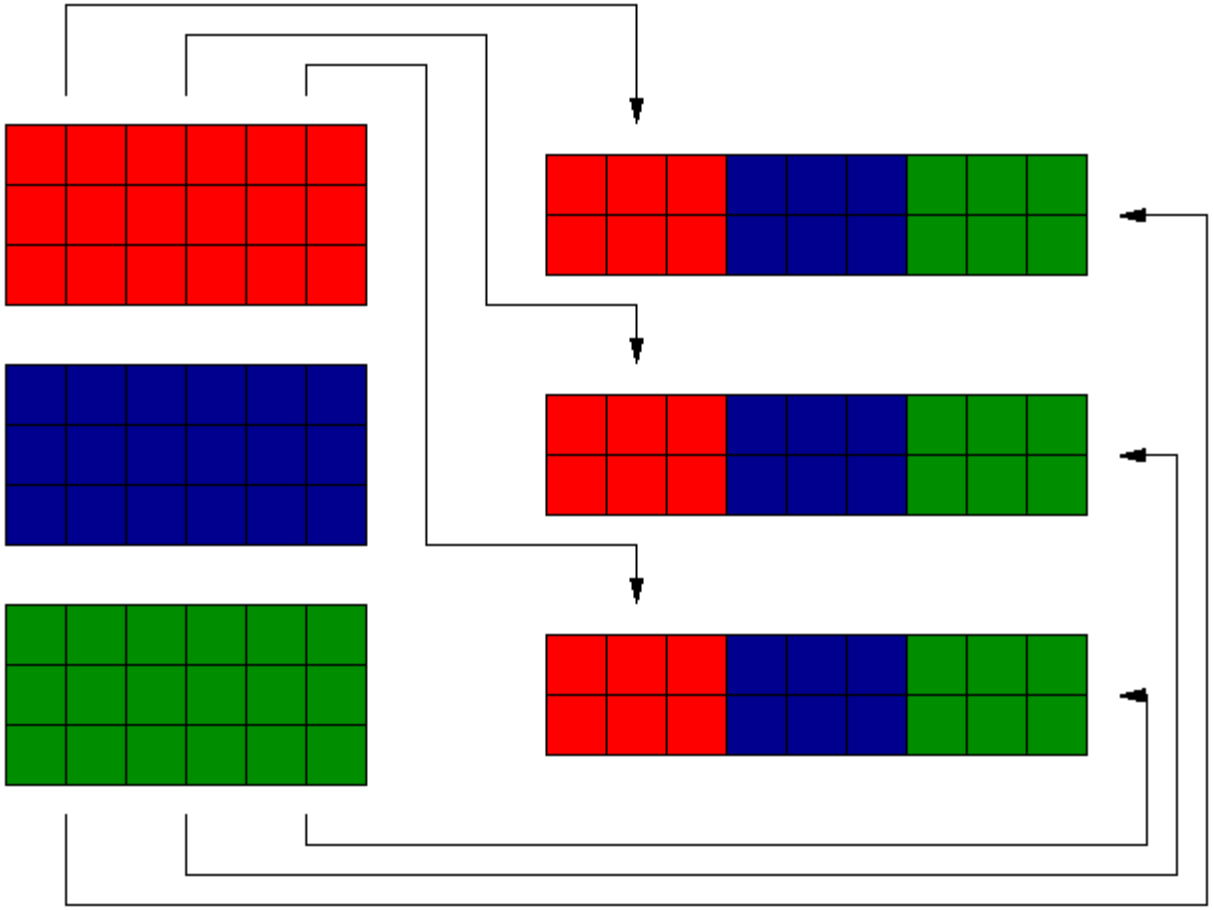
Distribute array onto 1D processor grid

- Example:
  - 6 × 9 array
  - 3 processors
  - Assuming row major order (C convention)
- Perform 1st FFT:
  - Each processor transforms 3 arrays of 6 elements
- What next?



Data transposition

- Divide up the array by columns for 2nd FFT
- Depending on FFT library simultaneous transpose can be advantageous (shown on figure)



Perform 2nd FFT

- What used to be the columns of the original array is now in row-major order ☺
- Do the 2nd FFT
  - In the example:
    - Each processor performs 2 FFTs of an array of length 9
- Rearrange data as required by following code
  - Examples:
    - Undoing the transpose
    - Redistributing data onto 2D grid
    - Sometimes: nothing needs done ☺

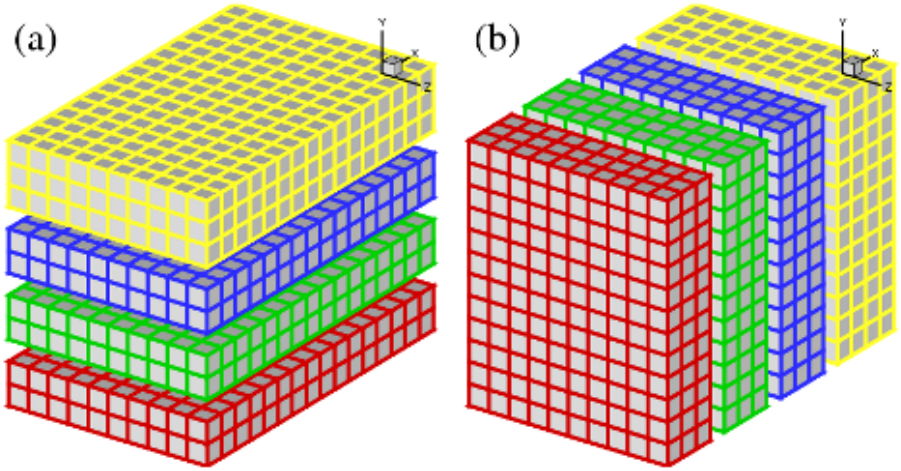
Fourier Transformation of a 3D array

- Definition of the Fourier Transformation of a 3D array $\boldsymbol{A_{x,y,z}}$

$$\tilde{A}_{u,v,w} := \underbrace{\underbrace{\underbrace{\sum_{x=0}^{L-1}\sum_{y=0}^{M-1}\sum_{z=0}^{N-1} A_{x,y,z}\exp(-2\pi\mathrm{i}\frac{wz}{N})}_{\text{1st 1D FT along } z}\exp(-2\pi\mathrm{i}\frac{vy}{M})}_{\text{2nd 1D FT along } y}\exp(-2\pi\mathrm{i}\frac{ux}{L})}_{\text{3rd 1D FT along } x}$$

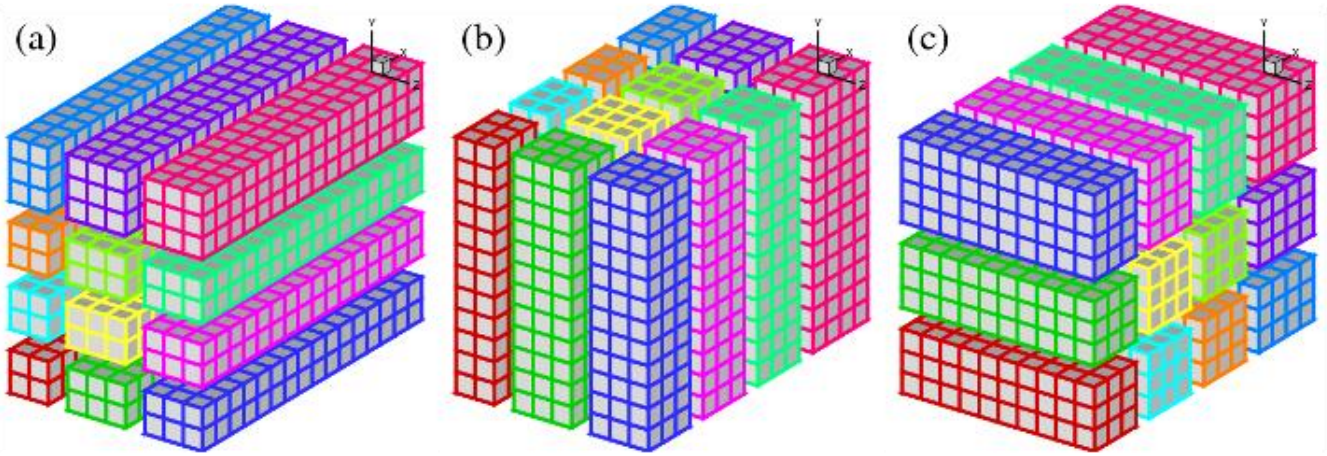
- Can be performed as three subsequent 1 dimensional FFTs

FFT Slab and Pencil decomposition



Slab: decomp in Y and X directions using 4 cores

Pencil: (a) X-, (b) Y- and (c) Z-pencils using 12 cores



Images from 2DECOMP&FFT library http://www.2decomp.org/decomp.html

FFT: Slab vs Pencil

- Slab
  - Pros: Simple with moderate amount of inter-processor communication
  - Cons: Limited to N procs for N^3 data
- Pencil
  - Pros: faster on massively parallel supercomputers (i.e. lots of cores)
  - Cons: More communications and now more complicated
- Pencil generally better with high core count but not so good for larger arrays on moderate number of cores
- Note: FFTW only does slab!

Summary of FFTs

- Parallelisation of an individual 1D FFT is hard
  - Presently works best for large problems
  - Recent advances in algorithms & hardware encouraging
- Multidimensional problems need to calculate many 1D FFTs
  - Parallelisable by distributing entire FFTs onto the processors and using a standard serial 1D FFT library
  - Requires redistributing the data between FFT dimensions
  - Need to think about decomposition (e.g. slab vs pencil)
- FFT can be used to reduce computational complexity of Fourier transform calculations from $O(n^2)$ to $O(n \log(n))$
  - Applications in signal processing, CFD, probability, etc.