RESEARCH
COUNCILS UK

nag®

HECToR

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Libraries I

David Quigley

Data storage in FORTRAN and C, cache based architectures and BLAS.

Libraries I

D. Quigley

Department of Physics
University of Warwick

Plan

1. Array handling in Fortran and C.
2. Cache architectures and performance.
3. Using BLAS for efficiency.

Array handling in Fortran and C.

Introduction

The practicalities of using libraries often involve calling C routines from Fortran or vice versa.

Even if a wrapper interface is provided we still need to understand how our data is stored and passed between functions/subroutines.

Some knowledge of what is happening "behind the scenes" is required here.

We will also examine issues controlling performance of array operations, and introduce the BLAS as a means of maximising performance.

Storing 2D matrices

Recall previous lecture.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

Arrays in Fortran are one-based, arrays in C are zero-based.

Fortran :

```
integer, dimension(3,3) :: A
```

```
A(2,3) = 8 ; A(3,1) = 3
```

C :

```
int A[3][3];
```

```
A[1][2] = 8 ; A[2][0] = 3;
```


Storing 2D matrices

Computer memory is linear, the location of each data word is specified by a single coordinate (base address + offset).

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}$$

This is *column-major* ordering. Each column is placed head-to-toe in a line.

Fortran stores multidimensional arrays in this format.

Computer memory →

	1	2	3	4	5	6	7	8	9			

Storing 2D matrices

Computer memory is linear, the location of each data word is specified by a single coordinate (base address + offset).

This is row-major ordering. Each row is placed end-to-end in a line.

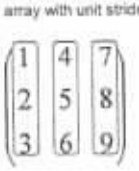
C stores multidimensional arrays in this format.

Computer memory →

1	4	7	2	5	8	3	6	9			

Storing 2D matrices

In Fortran the left-most index varies most rapidly, here we write the array with unit stride.




```
k = 1
do j = 1,3
  do i = 1,3
     A(i,j) = k
     k = k + 1
  end do
end do
```

Computer memory →

1	2	3	4	5	6	7	8	9			

Storing 2D matrices

In C/C++ the right-most index varies most rapidly. Hence if we keep the same indexing we write the array with stride 3.


```
k = 1;
for (i=0;i<3;i++){
    for (j=0;j<3;j++){
        A[i][j] = k;
        k++;
    }
}
```

Computer memory →

1	4	7	2	5	8	3	6	9			

Passing 2D matrices - C

Be VERY careful when passing 2D arrays in C.

We normally pass arrays by passing a pointer to the first storage element, and then passing the array dimensions as separate arguments.

Many library functions will expect to receive a pointer to a float (single precision) or double (double precision).

Depending on how we create the array there are many ways of doing this.

Passing 2D matrices - C

Computer memory →

1	4	7	2	5	8	3	6	9			

`int A[3][3];`	Declares a static 2 dimensional array.
`A[0][0]`	Is the value "1".
`A[2]`	Is a pointer to the memory location containing "3".
`*A[1]`	Is the value "2".
`*A`	Is a pointer to the first element of the array.
`A`	Is a pointer to the first element of the array.
`&A[1][2]`	Is a pointer to the memory location containing "6".

Passing 2D matrices - C

Computer memory (only if we are *very* lucky) →

1	4	7	2	5	8	3	6	9			

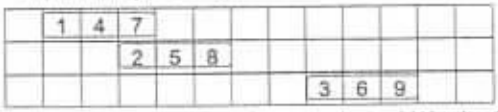
No guarantee that storage is contiguous – each row might be placed anywhere.

```
int **A;
A = (int **)malloc(3*sizeof(int *));
for (i=0;i<3;i++) {
    A[i] = (int *)malloc(3*sizeof(int));
}
```

Dynamic allocation of storage. A is an array of pointers, each of which points to the start of a row of 3 ints.

Passing 2D matrices - C

Computer memory (more likely) ⟶



No guarantee that storage is contiguous – each row might be placed anywhere.

```
int **A;
A = (int **)malloc(3*sizeof(int *));
for (i=0;i<3;i++) {
    A[i] = (int *)malloc(3*sizeof(int));
}
```

Dynamic allocation of storage. A is an array of pointers, each of which points to the start of a row of 3 ints.

Passing 2D matrices - C

```
int **A;
A = (int **)malloc(3*sizeof(int *));
for (i=0;i<3;i++) {
    A[i] = (int *)malloc(3*sizeof(int));
}
```

Dynamic allocation of storage. A is an array of pointers, each of which points to the start of a row of 3 ints.

`A[0][0]`	Is the value "1".
`A[2]`	Is a pointer to the memory location containing "3".
`*A[1]`	Is the value "2".
`A[0]`	Is a pointer to the first element of the array.
`*A`	Is a pointer to the first element of the array.
`&A[0][0]`	Is a pointer to the first element of the array.
`A`	Is a pointer *to a pointer* to the first element.

Array Passing in Fortran

Fortran programmers are probably unfamiliar with passing by reference vs by value.

The Fortran standard does *not* define what happens behind the scenes when passing between subprograms. Most implementations pass everything by reference and store multidimensional arrays contiguously, but this is not guaranteed to be the case for all compilers!

Fortran 2003 has C interoperability built in, but many standard libraries have yet to be upgraded to this new standard (or even Fortran 90 for that matter).

We can pass the size of the array as separate variables (recommended) or assume the array has the same size/shape as in the calling subroutine/ program.

Mixing languages

We can pass our matrix from C to Fortran.

```
// prototype
void myfortranmxb_(int *N, int *A);

int A[3][3];  // static declaration

k = 1;
for (j=0;j<N;j++){
    for (i=0;i<N;i++){
        A[i][j] = k;
        k++;
    }
}
myfortranmxb_(&N,&A[0][0]);
```

```
subroutine myfortranmxb(N,A)
  integer,intent(in)    :: N
  integer,dimension(N,N) :: A

  do j = 1,N
    do i = 1,N
      print*,A(i,j)
    end do
  end do
end subroutine myfortranmxb
```


Mixing languages (F2003)

We can pass our matrix from C to Fortran.

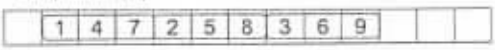
```
// prototype
void myfortranmxb(int N, int *A);

int A[3][3];  // static declaration

k = 1;
for (j=0;j<N;j++){
    for (i=0;i<N;i++){
        A[i][j] = k;
        k++;
    }
}
myfortranmxb(N,&A[0][0]);
```

```
! Must be contained in a module
use iso_c_binding

subroutine myfortranmxb(N,A) bind(c)
  integer(c_int),value          :: N
  integer(c_int),dimension(N,N) :: A

  do j = 1,N
    do i = 1,N
      print*,A(i,j)
    end do
  end do
end subroutine myfortranmxb
```


Mixing languages

C has stored our matrix A as three rows:

Computer memory ⟶



which Fortran will interpret as the 3 columns of B:

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}^T = A^T \neq A$$

Implications

2D arrays in C can be a pain. It is often better to use a 1D array explicitly and sacrifice the convenience of direct indexing, especially if we need to call Fortran routines.

```
int *C;
C=(int *)malloc(N*M*sizeof(int));
for (j=0;j<M;j++){
    for (i=0;i<N;i++){
        C[i+N*j] = A[i][j];
    }
}
myfortransub_(&N,C);
```

Note transposition into column-major. Fortran is now happy!

Cache architectures and performance

Memory Hierarchy

Registers:
- Stores data CPU is currently operating on.
- ~32 registers per CPU core.

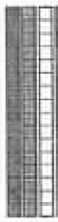
L1 cache:
- Small (e.g. 32 Kb) on CPU, fast SRAM.
- Takes 1-3 clock cycles to serve a memory request.

L2 cache:
- Larger (e.g. 4 Mb) usually also on CPU.
- Takes 5-25 clock cycles to serve a memory request.

Main Memory (e.g. 2 Gb)
- Takes 30-300 clock cycles to serve a memory request.

Substantial gains in performance by minimising number of reads/ writes to main memory.

Direct Mapping

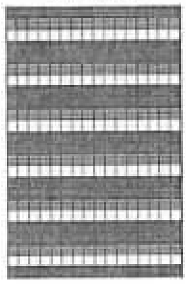


Whenever a value is read from main memory, an **entire** ***cache line*** is populated from memory.

Data already on that cache line is erased (or written back to memory in write-back vs write-through caches).

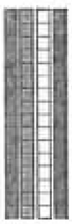
Cache memory

Main memory



e.g. 4 cache *lines* each holding 16 64-bit words (one 64 word = 1 double precision number)

Direct Mapping

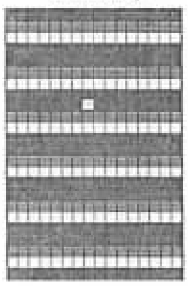


Whenever a value is read from main memory, an **entire** ***cache line*** is populated from memory.

Data already on that cache line is erased (or written back to memory in write-back vs write-through caches).

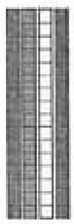
Cache memory

Main memory



e.g. 4 cache *lines* each holding 16 64-bit words (one 64 word = 1 double precision number)

Direct Mapping

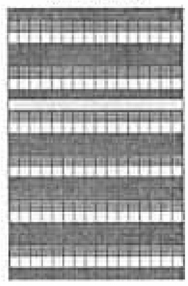


Whenever a value is read from main memory, an **entire** ***cache line*** is populated from memory.

Data already on that cache line is erased (or written back to memory in write-back vs write-through caches).

Cache memory

Main memory



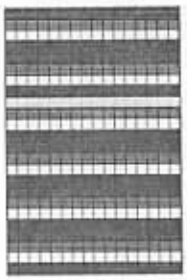
e.g. 4 cache *lines* each holding 16 64-bit words (one 64 word = 1 double precision number)

Direct Mapping

Whenever a value is read from main memory, an entire *cache line* is populated from memory.

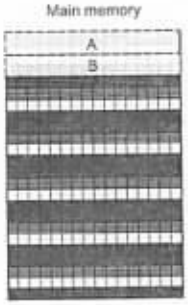
Data already on that cache line is erased (or written back to memory in write-back vs write-through caches).

Cache memory



e.g. 4 cache *lines* each holding 16 64-bit words (one 64 word = 1 double precision number)

Happy Cache



A and B map onto *different* cache lines.

2 cache lines in use at all times.

4 reads from main memory.

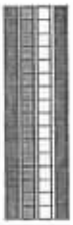
Cache Thrashing

A_i and B_i always map onto the same cache line.

Only using 1 cache line at a time.

128 reads from main memory c.f. 4 reads for problem of half the size.

Padding?



A_i and B_i now map onto different cache lines.

2 cache lines in use at a time.

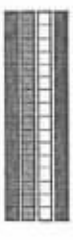
8 reads from main memory c.f. 4 reads for problem of half the size – much better.

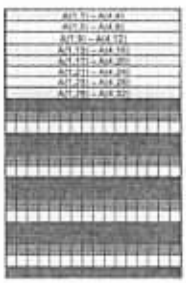
2d data – Fortran

Each addition triggers load of a new cache line. 2 lines in use at a time.

128 reads from main memory total.

2d data – Fortran

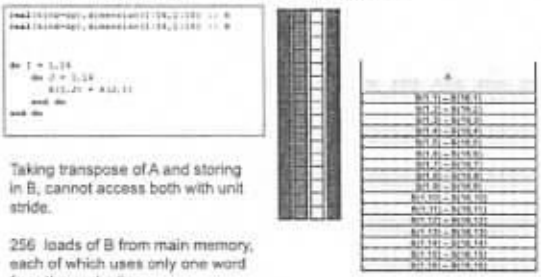




Now stepping through memory with unit stride.

8 loads from main memory total.

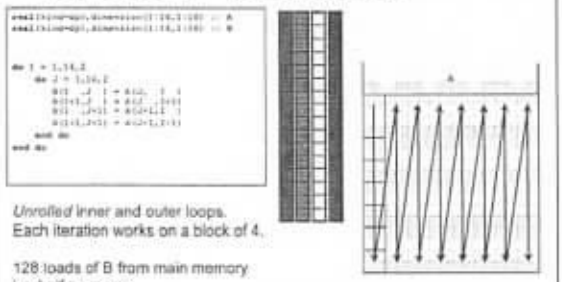
Blocking – an example



Taking transpose of A and storing in B, cannot access both with unit stride.

256 loads of B from main memory, each of which uses only one word from the cache line.

Blocking – an example



Unrolled inner and outer loops. Each iteration works on a block of 4.

128 loads of B from main memory i.e. half as many.

Some wisdom

"There are two rules of code optimisation:
Rule 1: Don't do it!
Rule 2 (for experts only): Don't do it yet!"
- M.A. Jackson

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity."
- W.A. Wulf

It would be hugely beneficial if experts had already written a set of routines for efficiently performing array operations in a cache-friendly way. Then we wouldn't have to worry about it ourselves.

They have – the "Basic Linear Algebra Subprograms" or BLAS.

BLAS

BLAS - http://www.netlib.org/blas/

Set of Fortran routines for array operations.

The reference implementation is "tuned" by hardware vendors to match cache architectures and other characteristics.

Vendor provided BLAS can be found within (for example):

- MKL – Intel Math Kernel Library.
- ACML – AMD Core Math Library.
- ESSL – IBM Engineering and Scientific Subroutine Library.

See also ATLAS and other self-tuning implementations.

BLAS Levels

Level 1
Vector operations, e.g. $y = \alpha x + y$

Level 2
Matrix-vector operations, e.g. $y = \alpha A x + \beta y$

Level 3
Matrix-matrix operations, e.g. $C = \alpha AB + \beta C$

BLAS routines implement several variants on each routine, depending on precision and any special matrix type.

BLAS Naming Conventions

<character> + <name> + <mod>

Character	Name	Mod
Specifies precision	Specifies operation (level 1) or matrix storage (levels 2&3) e.g.	Additional details e.g.
s = single d = double c = complex z = double complex	axpy = vector axpy axby = scalar a.x plus y ge = general matrix sp = symmetric matrix	c = conjugated vector mv = matrix-vector product mm = matrix-matrix product

e.g. scopy – copies a single precision vector
dgemm – double precision matrix-matrix product

Too many to list here, consult the documentation, i.e. RTLM!

Vector Arguments (Fortran)

BLAS allows us to be flexible in how we use storage. As a result we must pass extra arguments that might seem redundant.

For example:
```
call dcopy(N,x,1,y,1)
```
Copies x into y

```
call dcopy(N/2,x(2),2,y,1)
```
Copies even elements of x into y

We pass the number of elements to process, and for each array the first element to process (actually a pointer behind the scenes) and the stride to use.

Matrix Arguments (Fortran)

Similarly we specify extra arguments when passing matrixes

For example:
```
call dgemm(transA,M,N,alpha,A,ldA,X,incX,beta,Y,incY)
```

$$y = \alpha Ax + \beta y$$

Extra matrix arguments for A:

TransA	ldA
Extra operation to apply to A: 'n' = no operation 't' = transpose (useful for C) 'c' = conjugate (if relevant)	"Leading dimension of A" Number of storage elements between start of each column (Fortran) or row (C).

CBLAS Interface

A standard set of C wrapper functions for BLAS. Can be found in most vendor-BLAS implementations and in the GNU Scientific Library (GSL).

Fortran routine names are prefixed with "cblas_" e.g.

```
// Function prototype for CBLAS interface to dnrm2
double cblas_dnrm2(const int N, const double *X, const int incX);
```

Fortan subroutines (rather than functions) do not have a return value and so are declared as void functions.

All prototypes can be found in the header file for your implementation, or in the documentation.

CBLAS Interface

C did not have a complex type historically. BLAS / CBLAS will expect to find complex data as arranged in Fortran, i.e. interleaved real & imaginary components.

We can do this using a struct (with a health warning):

```
// Single precision complex
struct scomplex { float re; float im; };

// Double precision complex
struct dcomplex { double re; double im;};

// Double precision complex vector
struct dcomplex a[N];
```

Warning: re, and im are not guaranteed to be contiguous in memory!
(but usually are)

The CBLAS interface to Fortran functions which return complex data have an extra argument appended – this is a pointer to the complex array where the result should be stored.

CBLAS Interface

C99 introduced built-in complex types to the standard library.

```
#include <complex.h>

// double precision complex array
double complex a;

a = 2.0 + 3.0*I;

printf("a = %f + %f i\n",creal(a),cimag(a));

double *b = (double *)&a;

printf("a = %f + %f i\n",b[0],b[1]);
```

The double complex type is binary compatible with a two-element array containing the real and complex types, i.e. it is Fortran-happy!

CBLAS Interface

Matrix character arguments are dealt with using predefined constants, with an extra CBLAS argument which specifies if our data is in row-major or column-major format e.g.

```
cblas_dgemv(CblasRowMajor, CblasNoTrans, M, N, alpha, aa[0][0], lda,
            xx[0], incX, beta, yy[0], incY);
```

Note that M, N, lda etc are passed by value.

1st argument could be CblasRowMajor or CblasColMajor

2nd argument could be CblasTrans, CblasConjTrans or CblasNoTrans

Getting Hands-On

Best way to learn is to get stuck in!

This afternoon we will:

- Become familiar with CSC Linux.
- Gain experience in using libraries.
- Work with complex numbers in BLAS.
- Test the performance of BLAS.