

High Performance Computing

James Fergusson



Contents

1	FORTRAN	7
1.1	Background	7
1.2	FORTRAN77	7
1.3	FORTRAN90	11
2	Compilation	19
2.1	Makefiles	19
2.2	CMake	21
2.3	Helper Tools	24
3	Introduction to HPC	31
4	Optimisation	37



Preamble

Details:

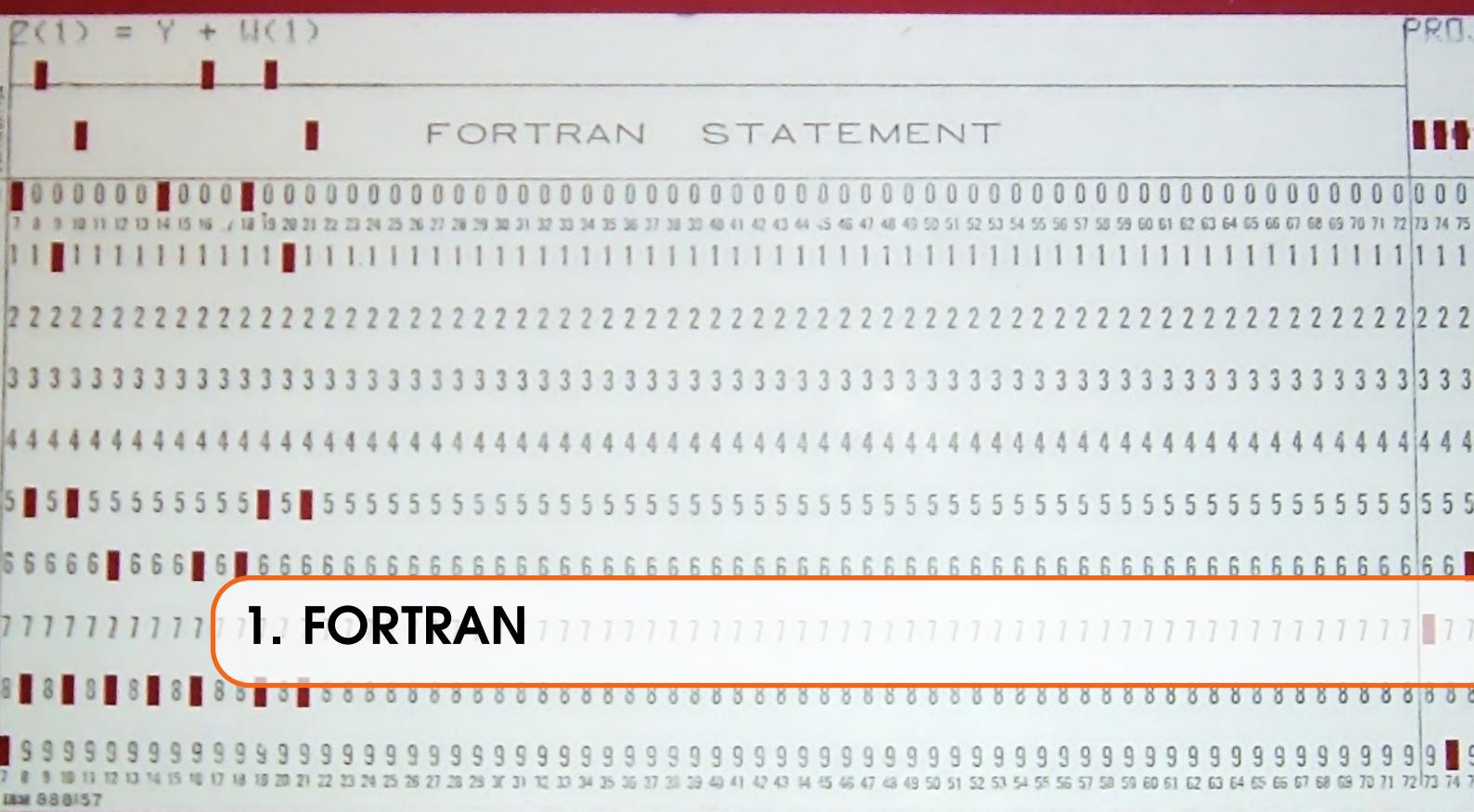
Dr James Fergusson

Room: B1:12

J.Fergusson@DAMTP.cam.ac.uk

Textbooks:

??



1. FORTRAN

1.1 Background

FORTRAN is one of the oldest coding languages. It was first originally developed in the 50s and was designed to work with punch cards which influences the standards created for it. So why would we want to learn about such an old fashioned language, surely we should just switch to more modern languages that are widely used like C++. Well the reason is that FORTRAN was created specifically for numerical computation (FORTRAN is a abbreviation of FORmular TRANslator) when memory and compute were very limited. This means that there is a large pool of legacy code for standard numerical tasks all of which is written in FORTRAN. In fact many standard libraries used in C++, Python and other languages are wrapped FORTRAN code (for example numpy). It is also likely that you will need to interact with FORTRAN codes in academia as many numerical codes used for scientific tasks use it. For example, while C++ dominates software development current statistics for ARCHER2 (a key supercomputer for scientific research) use are:

1. FORTRAN - 81.1%
2. C++ - 14.9%
3. Python - 2.1%
4. Other - 1.9%

so FORTRAN is still dominant in academic circles.

1.2 FORTRAN77

The first FORTRAN standard was released in 1966 but wasn't widely adopted until its revision in 1977. This standard, called FORTRAN77, became very widely used, and persisted for 13 years. It is now obsolete for new development but there is a large set of legacy code still used so you may come across it occasionally. It is quite tricky as it has several features which are significantly different to modern coding languages. We will briefly review its quirks so you can recognise it when you see it (and understand why all of your edits make it crash)

Firstly, it was based on punchcards so had some very specific requirement for formating. Each line could only be 80 characters long (as this was the width of a punchcard). Each line reserved

specific purpose for each column:

1. Column 1 : Any character here indicates the line is a comment (normal to use "C" or "*")
2. Column 2-5 : Label space, used for "GO TO" commands and "FORMAT" specifiers
3. Column 6 : Any character here indicates the line is a continuation of the previous line (normal to use "+")
4. Column 7-72 : Programming Statements
5. Column 73-80 : Sequence Number used to reorder punchcards if they were dropped

So the first issue with working with FORTRAN77 is you have to spend a bit of time counting spaces. If you accidentally put characters in the first spaces your line is ignored as a comment, in the first 5 the initial characters are interpreted as a label. Anything in space 6 appends this line to the one above. Anything that comes after character 72 is ignored.

Here is a simple Fortran77 program (it is typical to use the extension .f or .for for FORTRAN77):

Snippets/F77/HelloWorld.f

```

1      PROGRAM main
2 c      Simple hello world program
3      WRITE(*,*) 'Hello World'
4      STOP
5      END

```

We should note the following points:

1. You start your code with the command `PROGRAM` and end them with `END`.
2. To write output to the terminal you use `WRITE(*,*)` where the first * denotes the terminal and the second denotes unspecified formatting.
3. You have to `STOP` the program before it `ENDS`

To compile and run this code we use the gfortran compiler with the commands:

```

1 $ gfortran HelloWorld.f -o helloworld.exe
2 $ ./helloworld.exe
3 Hello World
4 $

```

Due to the limited row length, indentation is sometimes skipped which can make it hard to read. Also FORTRAN ignores all white space, which some programmes use to optimise line lengths making this even worse.

Here are the intrinsic variables for FORTRAN77:

Snippets/F77/Intrinsics.f

```

1 PROGRAM main
2 IMPLICIT NONE
3
4 INTEGER i,j,k
5 REAL x,y,z1,z2,z3,z4
6 DOUBLE PRECISION d1, d2
7 COMPLEX c1
8 LOGICAL bool1, bool2, bool3, bool4
9 CHARACTER str*8
10 REAL arr(4,5,6)
11
12 i = 2
13 j = 3

```

```

14 x = 2.0
15 y = 3.0
16 c1 = (1,3)
17 str = 'hello'
18
19 z1 = x+y
20 z2 = x*y
21 z3 = y/x
22 z4 = y**x
23 WRITE(*,*) z1,z2,z3,z4
24
25 bool1 = i .LT. j
26 bool2 = i .GE. j
27 bool3 = i .EQ. j
28 bool4 = i .NE. j
29 WRITE(*,*) bool1,bool2,bool3,bool4
30
31 STOP
32 END

```

Firstly I have put the command `INTRINSIC NONE` (technically not part of the 77 standard but widely supported in 77 compilers) which switched off a standard FORTRAN feature which is that all undefined variables that start with a letter i-n are automatically assigned as `INTEGER` and all others are assigned as `REAL`. All variable names (this includes functions and subroutines) can only be 6 characters long and must start with a letter. This is why in numerical packages you often see names like: `POTRF` for Choleski decomposition in the LAPACK numerical library (see more examples here: <https://icl.utk.edu/~mgates3/docs/lapack.html>).

In FORTRAN77 we only have one integer, which is 32 bit, but we have both 32 (`REAL`) and 64 (`DOUBLE PRECISION`) bit floating point numbers. We have booleans `LOGICAL` and strings with length defined by the *num part. Conveniently we have complex numbers as an intrinsic data type unlike other languages. Arrays are intrinsic, but not dynamic, in FORTRAN77. You can create multi-dimension arrays (up to 7 dimensions) simply by listing the dimension sizes in brackets when declaring them. Finally all declaration (ie assigning variables) must happen *before* any statements (code that does stuff with them). You cannot add variables halfway though some code. This means that you will invariably spend a lot of time scrolling up and down the code when working on it.

Control flow in FORTRAN77 is more limited intrinsically but can be extended to any case by using `GOTO` instructions (which can make your code very hard to read)

Snippets/F77/Controlflow.f

```

1      PROGRAM main
2      IMPLICIT NONE
3
4      INTEGER i,j,k
5      INTEGER n,m
6
7      n=0
8      m=1
9
10     IF (n .EQ. m) THEN
11         WRITE(*,*) "same"
12     ELSE IF (n .GT. m)
13         WRITE(*,*) "larger"
14     ELSE

```

```

15      WRITE(*,*) "smaller"
16  END IF
17
18  DO 99 i=1,10
19      n = n+i
20 99  CONTINUE
21  WRITE(*,*) n
22
23 88  IF(m .GT. 10) THEN
24      WRITE(*,*) m
25      m = m+1
26      GOTO 88
27  END IF
28  WRITE(*,*) m
29
30  STOP
31  END

```

Here we see the **IF** and **DO** structures plus a **IF+GOTO** which defines a 'while' loop. Finally, as FORTRAN does not care about spaces or case (technically the FORTRAN standard assumes all uppercase but most compilers don't care) **END IF GO TO** are identical to **endif** and **goto**. I personally always write FORTRAN instructions in caps, and variables in lower case, as I think it looks better but this choice is personal.

Fortran77 allows two subprogrammes, **FUNCTION** and **SUBROUTINE**. The first is a normal function $f(x,y,\dots)$ and the second takes multiple parameters and modifies them.

Snippets/F77/Subprograms.f

```

1 PROGRAM main
2 IMPLICIT NONE
3
4 INTEGER i,j,k
5 INTEGER add
6
7 i=1, j=2
8 k = add(i,j)
9 WRITE(*,*) k
10
11 CALL rotate(i,j,k)
12 WRITE(*,*) i,j,k
13
14 STOP
15 END
16
17 INTEGER FUNCTION add(i,j)
18 IMPLICIT NONE
19 INTEGER i,j
20 add = i+j
21 RETURN
22 END
23
24 SUBROUTINE rotate(i,j,k)
25 IMPLICIT NONE
26 INTEGER i,j,k,n
27 n=i
28 i=j
29 j=k

```

```

30 k=n
31 RETURN
32 END

```

Here the routines are just listed at the end after the main **PROGRAM** statements. If the code gets too long you can place subprograms in separate files and add them with the statement **INCLUDE filename.f** which simply cuts and pastes the code from the file to this location in the main program. Key points are that the function name needs to have a type in both the main program and its declaration, also the name is the variable that you return. Subroutines operate on all variables passed to them and require a **CALL** statement. Remember that implicit types apply to these too if **IMPLICIT NONE** is not set, and the 6 character limit still applies to names.

There is quite a bit more I could put here but as you should only use FORTRAN77 if you really have to we will move onto the much more common, and modern, FORTRAN90 standard which is the most common version used today

1.3 FORTRAN90

Fortran90 was a major revision of the standard. So much so that the two standards, while backwards compatible, cannot be used in the same file with each other so a small amount of care must be taken if you wish to combine them.

The main differences between FORTRAN90 and FORTRAN77 are:

1. Dropped the static formatting requirement and adopted the flexible format that all modern codes follow. Now variable names can be 31 characters long and lines 132 characters long. Comments start with, lines are continued by putting & at the end of the line to be continued and lower case letters are formally allowed (but FORTRAN is still case insensitive).
2. Introduced **MODULEs** to replace the **INCLUDE** method for functions and subroutines.
3. Allowed dynamic array allocation and introduced intrinsic array functions.
4. New control structures like **DO WHILE** to avoid the need for **GOTO** and no longer require labels.

Let us look at the intrinsics again as there are some subtleties. We still have the 5 standard intrinsic types: **INTEGER**, **REAL**, **COMPLEX**, **LOGICAL**, and **CHARACTER** but we can control the precision of each with **KIND**

Snippets/F90/Intrinsics.f90

```

1 PROGRAM main
2   IMPLICIT NONE
3
4   INTEGER      :: i
5   REAL         :: x
6   COMPLEX      :: c
7   LOGICAL      :: bool
8   CHARACTER    :: a
9   CHARACTER(len=180) :: string
10
11 ! values can be assigned at creation (could do this in 77 but it was odd)
12 REAL          :: y=3.5
13 REAL, PARAMETER :: pi=3.1415927
14
15 ! change allocation size
16 INTEGER(KIND=2) :: int_short
17 INTEGER(KIND=4) :: int_normal
18 INTEGER(KIND=8) :: int_long ! not always available

```

```

19      REAL(KIND=4) :: real_normal
20      REAL(KIND=8) :: real_long ! also DOUBLE PRECISION
21      REAL(KIND=16) :: real_quad ! not always available
22
23
24      COMPLEX(KIND=4) :: cmplx_normal
25      COMPLEX(KIND=8) :: cmplx_long
26
27 END PROGRAM

```

Arrays are where FORTRAN90 shines over C++. They are an intrinsic type and come with multiple tools for their manipulation. Python users will find this all quite familiar, numpy is a lightly wrapped version of FORTRANs native array capabilities so most of what you can do in python is available here.

Snippets/F90/arrays.f90

```

1 PROGRAM main
2   IMPLICIT NONE
3
4   ! Fixed size at compile
5   INTEGER i,j,k
6   INTEGER           :: vector1(5)
7   INTEGER           :: array1(5,5)
8   INTEGER, DIMENSION(5,5) :: array2
9
10  ! Dynamic allocation
11  INTEGER, ALLOCATABLE          :: vector2(:)
12  INTEGER, ALLOCATABLE          :: array3(:,:)
13  INTEGER, DIMENSION(:,:,:), ALLOCATABLE :: array4
14  INTEGER, POINTER             :: array5(:,:)
15  INTEGER, DIMENSION(:,:,:), POINTER    :: array6
16
17  ALLOCATE(array3(5,5))
18  IF (ALLOCATED(array3)) THEN
19    WRITE(*,*) 'Array 3 allocated'
20    WRITE(*,*) "array size", SIZE(array3,1), " X ", SIZE(array3,2)
21  END IF
22
23  ALLOCATE(array4(-5:3,0:4)) ! indicies can range from anything to anything else,
24  ! default is 3 -> 1,2,3
25
26  ALLOCATE(array5(5,5))
27  ALLOCATE(array6(3,3))
28
29  array1 = 0 ! whole array assignment
30  array1(2,2) = 7 ! element assignment
31  vector1 = (/1,2,3,4,5/) ! all elements in one go
32  vector1 = (/(i**2,i=1,5)/) ! via a constructor rule
33  array1(1,:) = 5 ! row assignment
34  array1(3,1:5:2) = 3 ! assignment with step size -> (3,1), (3,3), (3,5) = 3
35
36  array2 = 5*array1 + 6 ! supports whole array element-wise calcuations (most
37  ! functions too!)
38
39  array5 = 1
40  array6 = array5(2:4,2:4) ! assiging from sub-array

```

```

39      ! Intrinsic array operations
40
41      i = SUM(array1) ! add all elements together
42      j = PRODUCT(array1) ! multiply all elements together
43      k = MAXVAL(array1) ! find the maximum
44      k = MAXVAL(array1, MASK = (MOD(array1,2)>0)) ! find the maximum odd number
45      vector1 = MAXVAL(array1, DIM=1) ! find the maximum in each row
46      i = SIZE(SHAPE(array1))
47      ALLOCATE(vector2(i))
48      vector2 = MAXLOC(array1) ! find the location of the maximum
49
50      array5 = MATMUL(array1,array2)
51      array1 = TRANSPOSE(array1)
52
53      ! Cool syntax for conditional array modification
54      WHERE (array1/=0)
55          array1 = -array1
56      ELSEWHERE
57          array1 = 5
58      END WHERE
59
60      ! NEED TO DEALLOCATE ALLOCATABLE ARRAYS AFTER USE!!
61      DEALLOCATE(array3)
62
63 END PROGRAM

```

Some points to note are that arrays are allocatable, but this means that you need to deallocate them once you are finished to conserve memory. The pointer array is so you can pass them to functions in unallocated form. Constructors only work on 1D arrays so you need to use `RESHAPE` statement to make multi-dimension arrays from them. Not in the example above are functions like `ALL` which applies conditions to the entire array, and `SPREAD` (which is the command which allows broadcasting in python to work) copies one array to multiple dimensions of a higher dimension array.

I/O in Fortran is handled via `READ` and `WRITE` statements. Format statements have to be specified in advance using `FORMAT` and a label which can be used in `READ` and `WRITE` statements. Files are opened and closed with labels so you can specify which file you are using.

Snippets/F90/io.f90

```

1 PROGRAM main
2     IMPLICIT NONE
3
4     INTEGER :: i
5     REAL :: x
6     INTEGER :: fileunit
7     CHARACTER(180) :: filename
8     REAL, DIMENSION(:,:), ALLOCATABLE :: array1
9
10    fileunit = 99
11
12    i = 4000
13    x = 2.45e3
14    ALLOCATE(array1(10,10))
15    array1 = 5.67
16    filename = "test.out"
17

```

```

18 90101 FORMAT('The output is: ',I5,' ',E13.5E3)
19
20 OPEN(UNIT=fileunit, FILE=filename)
21 WRITE(UNIT=fileunit,FMT=90101) i,x
22 CLOSE(UNIT=fileunit)
23
24 ! for arrays use this:
25 OPEN(UNIT=fileunit, FILE=filename, STATUS='REPLACE', FORM='UNFORMATTED',
26      ACCESS='STREAM')
26 WRITE(fileunit,*) array1
27 CLOSE(fileunit)
28
29 OPEN(UNIT=fileunit, FILE=filename, STATUS='OLD', FORM='UNFORMATTED',
30      ACCESS='STREAM')
30 READ(fileunit,*) array1
31 CLOSE(fileunit)
32
33 END PROGRAM

```

The other major addition to FORTRAN90 was the inclusions of **MODULEs**. This allows us to make packages of functions and subroutines that can be used everywhere in the code. Let's create a couple of modules as an example

Snippets/F90/module1.f90

```

1 MODULE keep_count
2   IMPLICIT NONE
3
4   INTEGER, SAVE, PRIVATE :: count
5
6   PUBLIC get_count, increment
7
8   CONTAINS
9
10  FUNCTION get_count()
11    IMPLICIT NONE
12    INTEGER get_count
13    get_count = count
14  END FUNCTION get_count
15
16  SUBROUTINE increment(amount)
17    IMPLICIT NONE
18    INTEGER, INTENT(IN) :: amount
19    count = count+amount
20  END SUBROUTINE increment
21
22  SUBROUTINE reset()
23    IMPLICIT NONE
24    count = 0
25  END SUBROUTINE reset
26
27 END MODULE keep_count

```

Snippets/F90/module2.f90

```

1 MODULE constants
2   IMPLICIT NONE
3
4   SAVE
5   DOUBLE PRECISION, PARAMETER :: pi=3.1415927
6   DOUBLE PRECISION, PARAMETER :: e=2.71828
7   DOUBLE PRECISION, PARAMETER :: G=6.67430e-11
8   DOUBLE PRECISION, PARAMETER :: c=299792458
9   DOUBLE PRECISION, PARAMETER :: h=6.62607015e-34
10
11 END MODULE constants

```

Snippets/F90/usemodule.f90

```

1 PROGRAM main
2
3   USE keep_count
4   USE constants, ONLY : speed_of_light => c
5
6   IMPLICIT NONE
7
8   INTEGER :: i
9
10  CALL reset()
11
12  DO i=10,99
13    IF (MOD(i,7)==0) THEN
14      CALL increment(1)
15    END IF
16  END DO
17
18  WRITE(*,*) 'number of 2 digit numbers divisible by 7 is: ', get_count()
19  WRITE(*,*) 'speed of light is: ', speed_of_light
20
21 END PROGRAM

```

Looking at these we see that to use a module we just add it with the `USE` command at the begining and we can use the `ONLY` statement to specify which portion of the module we want to use and `=>` to rename them to avoid local name clashes. Variables declared above the `CONTAINS` statement are global to the functions and subroutines in the module and can be accessed in the main programme, unless they are specified `PRIVATE` which hides them from the main programme. All functions and subroutines must come after the contains statement. The `SAVE` statement ensures that the value of parameters are not forgotton when the module goes out of scope i.e. it could be used by routines in two different modules and we want the value to persist for all (doing this allows us to define a “global” variable in a way that is protected as it can only be accessed via the module). The other key point is that all dummy variables names in module subroutines statements should have the extra qualifier `INTENT` which can be one of `IN`, `OUT`, `INOUT`. The compiler uses these to optimise your code but they are not required. `IN` variables should be ones that the subroutine uses but does not modify, `OUT` are ones that the subroutine modifies for return without reference to their input value and `INOUT` are variables whose value is used as input and is modified for return. All other variables internal to the subroutine are private and do not require statements.

We have data structures in FORTRAN90 (which are the equivalent of structs in C). These are defined as new `TYPEs`:

Snippets/F90/types.f90

```

1 PROGRAM main
2   IMPLICIT NONE
3
4   TYPE :: my_type
5     INTEGER :: moons
6     REAL :: coord_x, coord_y
7     CHARACTER(30) :: name
8   END TYPE my_type
9
10  TYPE(my_type) :: planets(8)
11  REAL :: vector1(8)
12  REAL :: x
13
14  planets(1)%moons = 0
15  planets(1)%name = "Mercury"
16  planets(1)%coord_x = 58e6
17  planets(1)%coord_y = 0e0
18
19  planets(3) = (/1,152e6,0e0,"Earth"/)
20
21  vector1 = planets%coord_x
22  x = SUM(planets%moons)
23
24 END PROGRAM

```

We can overload operators for types we create using modules as follows:

Snippets/F90/overload.f90

```

1 MODULE overload
2   IMPLICIT NONE
3
4   TYPE fraction
5     INTEGER :: numerator
6     INTEGER :: denominator
7   END TYPE fraction
8
9   INTERFACE OPERATOR (+)
10    MODULE PROCEDURE frac_add
11  END INTERFACE
12
13  CONTAINS
14
15  SUBROUTINE reduce(frac1)
16    IMPLICIT NONE
17    TYPE(fraction), INTENT(INOUT) :: frac1
18    INTEGER :: a,b
19
20    a = frac1%numerator
21    b = frac1%denominator
22
23    DO WHILE (b/=0)
24      a = b
25      b = MOD(a,b)
26    END DO
27
28    frac1%numerator = frac1%numerator/a

```

```
29      frac1%denominator = frac1%denominator/a
30
31  END SUBROUTINE reduce
32
33  FUNCTION frac_add(frac1,frac2)
34    IMPLICIT NONE
35    TYPE(fraction), INTENT(IN) :: frac1, frac2 ! needed for overloading
36    TYPE(fraction) :: frac_add
37
38    frac_add%numerator = frac1%numerator * frac2%denominator + frac2%numerator
39    ↵   * frac1%denominator
40    frac_add%denominator = frac1%denominator * frac2%denominator
41
42    CALL reduce(frac_add)
43
44  END FUNCTION frac_add
45
45 END MODULE overload
```

Modules can be compiled by the simple command we used previously

```
1 $ gfortran module1.f90 module2.f90 runmodules.f90 -o executable.exe
2 $ ./executable.exe
```

The key part is that the modules are specified in order of their dependency. Modules dependencies must be strictly hierarchical, so you cannot have module1 using module2 and module2 using module1, and compiled in order from bottom to top. Compilers execute the files in order that they receive them so putting lower modules first ensures that the .mod files exist for later modules/programs that use them.



2. Compilation

2.1 Makefiles

Compilation of code is a bit fiddly to do from the command line every time, especially once the project becomes large with multiple source files. This has been solved by using programmes that handle the building of projects automatically. The simplest of these is to create a **Makefile**

Makefiles consist of a list of rules, usually for updating files when they depend on change, but they can be used more generally. The rules take the form (note we need tab rather than 4 spaces for indentation):

```
1 target ... : dependency ...
2   commands
3   ...
4   ...
```

```
1 test :
2   echo "Hello!"
```

```
1 test1 : test2
2   echo "Hello two!"
3
4 test2 :
5   echo "Hello one!"
```

This is OK but a bit verbose. Just like when writing python we are better to use variables to make the code simpler to understand. In make files variables are created with = and := signs. The values are accessed by `$(var)`. The first assignment = is implicit, which means that it doesn't expand the rhs immediately, the second := is explicit, in that it does expand it before assignment. The difference can be seen in the examples:

```

1 var1 = $(var2)
2 var2 = "hello"
3 echo $(var1)
4
5 var3 = "hello"
6 var3 := $(var3)

```

Here we don't expand var1 until we get to echo `$(var1)` so it doesn't matter that var2 isn't defined when we assign var1. With `:=` this would matter as we would try to expand `$(var2)` when creating var1 and it wouldn't exist. Conversely for var3 if the second assignment was implicit this would create an infinite loop that can't be expanded. Here `:=` works fine as we would expand it before assignment.

There is also `?=` which assigns the variable only if it has not previously been assigned and `+=` which will add another element to a list (which is specified just by spaces between variable names), ie:

```

1 var1 = one two three
2 var1 += four

```

We can also create pattern specific variables using:

1. `%` – This will match any non-empty string and can be used in any string object, but only once.
2. `$@` – The filename of the target
3. `$<` – The first filename of the dependency
4. `$^` – The filenames of all the dependencies

This allows us to set up generic rules for all files of a specific type, like object files which are always created from their c files, ie:

```

1 # compilers, flags and libraries
2 CC = gcc
3 CFLAGS := -g -O3 -xHost
4
5 # library packages you are useing
6 LIBS :=
7
8 # Directories for code objects, and librarys
9 OBJDIR := Objects
10 SRCDIR := Code
11 BINDIR := .
12
13 # source file(s) without suffix
14 CFILES = file1 file2 file3
15
16 PROGRAMS = program1 program2
17
18 #This says don't look for a file called program1 or program2
19 .PHONY := $(PROGRAMS)
20
21 program1 : $(CFILES:%= $(OBJDIR)/%.o) $(mainfile1:%= $(OBJDIR)/%.o)
22   $(CC) $(CFLAGS) -I$(SRCDIR) $^ -o $(BINDIR)/$@ $(LIBS)
23
24 program2 : $(CFILES:%= $(OBJDIR)/%.o) $(mainfile2:%= $(OBJDIR)/%.o)
25   $(CC) $(CFLAGS) -I$(SRCDIR) $^ -o $(BINDIR)/$@ $(LIBS)
26
27 # everything that ends in '.o' should be made from the same file with '.c' instead
28 $(OBJDIR)/%.o: $(SRCDIR)/%.c
29   $(CC) $(CFLAGS) -I$(SRCDIR) -c $< -o $@
30
31 clean :
32   rm $(OBJDIR)*.o

```

Makefiles can be a pain to create as the syntax is pretty unreadable and you generally don't create them enough to get really familiar with the process. Typically you get one that works then just cut and paste it everywhere you work then debug the inevitable issues that come up. This will allow you to automate code building and is generally ok for most small projects. However, if you move to a new system the Makefile will need to be updated with the correct compilers, flags and libraries etc...

2.2 CMake

The creation of Makefiles is sufficiently difficult that there exist tools to automate the process for you. The industry standard approach is CMake which will generate Makefiles automatically from some small script files. The real benefit of using tools like CMake are that you can use them to automate many other build management tasks like installing and building libraries or fetching them from git repositories, running testing frameworks and using continuous integration, and installing compiled executables.

The other main benefit is that it makes your code portable so you can jump between platforms and it will automatically discover the local compiler and libraries needed for building. It ends up working a bit like a 'package manager' for C++, like conda is for python, where it looks after all the system dependencies of your project for you and even allows you to change the generator used (Make by default, but there are many others) and to easily add things like parallel builds for large projects.

You will need to install CMake onto your system (`brew install cmake` or similar on Linux

or Windows). Once you have done this we can see a simple cmake build file for a "Hello World" program:

Snippets/CMake/HelloWorld/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.15...3.26)
2
3 project(
4     HelloWorld
5     VERSION 1.0
6     LANGUAGES CXX
7 )
8
9 add_executable(hello main.cpp)
```

The cmake build file must always be called `CMakeLists.txt` and exist in the project directory (wherever you would normally initialise git). The first line in it must always be the command `cmake_minimum_required()` which specified the minimum version of cmake required to build the project. This is needed so as cmake commands have evolved significantly since its first release in 2000. While all scripts are backwards compatible we will be wanting to use many features from "modern" cmake which is generally anything from 3.13 on. In the above we have specified a range from 3.15 to 3.26. Next you need to give your project a name and optionally a version and language. Finally we need to specify an executable we want to build and which cpp file contains the main program to build from.

The project can be built with:

```

1 $ cmake -S . -B build
2 $ cmake --build build
```

The first builds the project from the `CMakeLists.txt` in the source directory `"."` and builds the project in the directory "build" (which it will create if it does not exist). The second uses the build to create the executable in the build directory.

We can add libraries simply with the following commands:

Snippets/CMake/Library/CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.15...3.26)
2
3 project(library_example
4     VERSION 1.0
5     LANGUAGES CXX
6 )
7
8 # This would work, but wouldn't build a library:
9 # add_executable(test main.cpp harmonic.cpp)
10 # Instead do this:
11
12 add_library(harmonic_lib
13     STATIC
14     harmonic.cpp
15     harmonic.hpp
16 )
17
18 add_executable(test main.cpp)
```

```

19
20 target_link_libraries(test PRIVATE harmonic_lib)

```

Here we add the libraries with `add_libraries()` and then describe the dependance graph with the command `target_link_libraries()`. The format of cmake commands are all (target OPTIONS dependancies..) which mimics that of Make. We can list multiple (same level) dependancies on the same line but heiracial depencanceis should be input on multiple lines, e.g.

```

1 target_link_libraries(executable PRIVATE lib1)
2 target_link_libraries(lib1 PRIVATE lib2)
3 target_link_libraries(lib2 PRIVATE lib3)

```

We used two options to help us create our libaraies, `STATIC` and `PRIVATE`. the first is the type of library we are creating which can be `STATIC`, `SHARED`, and `MODULE`. The difference is `STATIC` are liked at compile time, `SHARED` are linked at runtime. The first will copy the library code add put it in your executable, the other is more efficent for libraries that multiple executables use as then we only need one copy. A simple rule for this is to use `STATIC` for libraries you built that only one file uses and `SHARED` for libraries that you build that multiple files use. `MODULE` is for shared libaraies that you don't link to, but instead load at runtime with a commnad like `dlopen()`. The options for linking are `PUBLIC`, `PRIVATE`, and `INTERFACE`. `PUBLIC` is for when things that link to the target need to be able to know about the dependant, `PRIVATE` is for when they don't and `INTERFACE` is for when we need to know header but not detail, for example when using header libraries, you need to specify them as `INTERFACE`. It is possible to make lib1 depend on lib2 AND lib2 depend on lib1 and cmake still work but if this happens in your code, you should have a long hard think about what you were trying to acheive.

The simple examples above are not really how you should structure your projects. It is much better to make then hieracial with the code in seperate folders. Despite what you will read in most tutorials on cmake, there is no standard way to do this (and everyone complains about it). You should just aim for something helps make you code more understandable by clearly reflecting its purpose and structure. The key difference is that you have to put `CMakeLists.txt` files in each folder. I have included my personal favourite approach as an example but you can choose others. I like to keep `.ccp` and `.hpp` files together which makes sense for most personal projects. If you are designing libraries to be used by others then you will want to split off the public `.hpp` files into an `include` directory. I have all code in a folder called `src` and libraries in subfolders off that. I place all binaries created in a folder called `bin`. I put the cmake commands for building libraries in the library sub directories and cmake commands for building executables in the `src` directory. The project folder then just have general cmake commands to set up the project. You should add both `bin` and `build` to `git.ignore` files. This format is fairly simple to understand but has the weakness (strength?) of being strictly heirarcical.

Cmake can do lots more. It can manage the use of external libraries like MPI or OpenMP using the `find_package()`:

```

1 find_package(OpenMP)
2 if(OpenMP_CXX_FOUND)
3   target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
4 endif()
5
6 find_package(MPI REQUIRED)
7 target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)

```

Eternal code can be included in your project with `fetch_content()`:

```

1 FetchContent_Declare(contentname
2   GIT_REPOSITORY https://github.com/google/content.git
3   GIT_TAG        703bd9caab50b139428cea1aaff9974ebbe5742e
4 )
5 FetchContent_MakeAvailable(contentname)

```

And, as we will see later, cmake can manage continuous integration tasks

2.3 Helper Tools

There are several useful helper tools you should use for your code to make it easier for others to use and understand. We will show the several of them here

Doxygen

Doxygen is a tool that creates documentation for your code automatically from the comments inside it for a wide variety of languages. First you have to install it (`brew install doxygen` or download binaries for Linux/Windows). Then you just need to create a configuration file in the project directory using

```
1 $ doxygen -g doxygen.config
```

This creates a long configuration file which defines doxygen's behaviour. To begin with you can ignore almost all of it other than changing the following:

```

1 PROJECT_NAME          = "Doxygen Test"
2 OUTPUT_DIRECTORY       = docs
3 EXTRACT_ALL           = YES
4 EXTRACT_PRIVATE        = YES
5 INPUT                 = src
6 RECURSIVE             = YES

```

Documentation is then generated by the simple command

```
1 $ doxygen doxygen.config
```

Which will create documentation in the `OUTPUT_DIRECTORY` in both html and latex format, with the latter having a `MAKEFILE` to compile the document. This will be almost blank for the example projects as I have not added documentation to them. This can be fixed easily by adding some comments above (or inside) each function. This is done easiest by using the Doxygen extension to VSCode. Then you can add comment blocks above each function by typing `/**` then hitting enter which will create blocks like:

```

1  /**
2  * @brief
3  *
4  * @param x
5  * @param y
6  * @return
7  */
8 double harmonic::harm_add(double x, double y)

```

To which you can add a description of the function after `@brief` and a description of each parameter after `@param`. Play with this and see how the documentation is updated.

Doxygen is very easy to add to a project being build by CMake with the command `doxygen_add_docs`. We will jump straight to the general case where we define a cmake module file to automate this. We need to create a directory `cmake` and include add the following file, `Doxygen.cmake`, to it:

```

1 function(Doxygen input output)
2   find_package(Doxygen)
3
4   if (NOT DOXYGEN_FOUND)
5     add_custom_target(doxygen COMMAND false
6       COMMENT "Doxygen not found!!"
7     )
8   return()
9 endif()
10
11 set(DOXYGEN_GENERATE_HTML YES)
12 set(DOXYGEN_HTML_OUTPUT ${PROJECT_SOURCE_DIR}/${output}/html)
13 set(DOXYGEN_GENERATE_LATEX YES)
14 set(DOXYGEN_LATEX_OUTPUT ${PROJECT_SOURCE_DIR}/${output}/latex)
15 set(DOXYGEN_EXTRACT_ALL YES)
16 set(DOXYGEN_EXTRACT_PRIVATE YES)
17 set(DOXYGEN_RECURSIVE YES)
18 set(DOXYGEN QUIET YES)
19
20 doxygen_add_docs(doxygen
21   ${PROJECT_SOURCE_DIR}/${input}
22   COMMENT "Generating documentation"
23 )
24
25 endfunction()

```

We then update our `CMakeLists.txt` to include the new function:

```

1 cmake_minimum_required(VERSION 3.15...3.26)
2
3 project(folder_example
4   VERSION 1.0
5   LANGUAGES CXX
6 )
7
8 list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")
9
10 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/bin)
11
12 include(Doxygen)
13 Doxygen(src docs)
14
15 add_subdirectory(src)

```

And documentation is created on the command line by specifying the target doxygen

```
1 cmake --build build -t doxygen
```

Enforcing style

You should also add tools that help enforce formatting for your code to ensure a consistent style, and some static analysis to catch simple programming errors. clang-format does the former, and clang-tidy does the latter. These are some of the most popular tools but there are many alternatives like: cpplint and cppcheck. You can also combine these tools with other static checkers like include-what-you-use which removes unnecessary include statements.

clang-format

To use clang_format you just call it on the command line for whatever files you want to format:

```
1 clang-format -i --style=Google filename1.cpp
```

Where the `-i` means ‘inplace’, so re-format the file and put output to original filename, and `--style` defined the formatting style you want to use from LLVM, Google, Chromium, Mozilla, Webkit, Microsoft, GNU, custom where custom is defined by a file you create. We can automate this in cmake by adding a function like we did before. We need to include a ClangFormat.cmake file:

```

1 function(ClangFormat target directory)
2   find_program(CLANG-FORMAT_PATH clang-format REQUIRED)
3
4   # define a list of suffixes for files ot be considered
5   set(EXPRESSION h hpp hh c cc cxx cpp)
6   # prepend the directory to the list
7   list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")
8
9   # create list of files to format recuresivly and following symboloc links
10  file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
11    LIST_DIRECTORIES false ${EXPRESSION})
12 )
13
14 # Add command to run clang-format on list of files before build
15 add_custom_command(TARGET ${target} PRE_BUILD COMMAND
16   ${CLANG-FORMAT_PATH} -i --style=Google ${SOURCE_FILES})
17 )
18 endfunction()

```

Then we need to add the following lines to the CMakeLists.txt in the project and src directorys:

```

1 # in project one
2 include(Doxygen)
3 include(ClangFormat)
4
5 # in src one:
6 add_executable(main main.cpp)
7 ClangFormat(main .)

```

clang-tidy

Clang tidy also works on the command line, just like clang-format, on a file by file basis:

```
1 $ clang-tidy filename1.cpp -options...
```

We will again automate it with cmake in a similar fashion. First we create a function in a ClangTidy.cmake file:

```

1 function(ClangTidy target)
2   find_program(CLANG-TIDY_PATH clang-tidy REQUIRED)
3
4   set_target_properties(${target}
5     PROPERTIES CXX_CLANG_TIDY
6     "${CLANG-TIDY_PATH};-checks=clang-analyzer-*,"
7     bugprone-*,cppcoreguidelines-*,modernize-*,readability-*)
8   )
9 endfunction()

```

analyzer and bugprone are pretty good the next threee are a bit opinionated so can be neglected. Then we include it and add the function to the relevant CMakeLists.txt files (it's not recursive on libraries so you have to add it everywhere):

```

1 # in project one
2 include(Doxygen)
3 include(ClangFormat)
4 include(ClangTidy)
5
6 # in src one:
7 add_executable(main main.cpp)
8 ClangTidy(main)
9 ClangFormat(main .)
10
11 # in src/harmonic one:
12 add_library(harmonic_lib
13 ...
14 )
15 ClangTidy(harmonic_lib)

```

You have two other options you can add, `--warnings-as-errors=*` which will cause the build to fail if there are any warnings (for those who want to be hardcore about it) and `--fix` which will make automated fixes for issues it finds (beware this is a little buggy)

You should also consider building debug versions or your project with:

```

1 $ cmake -S . -B build/debug -DCMAKE_BUILD_TYPE=Debug
2 $ cmake -S . -B build/release -DCMAKE_BUILD_TYPE=Release
3 $ cmake --build build/debug
4 $ cmake --build build/release

```

Testing

The mainenance of code over time is a very challenging task. In the begining everything is easy, particularly if you are working on a project solo. However as time passes and the code evolves and expands or new developers are added it becomes an increasingly difficult task to know if or where bugs are being introduced to the project. Some may not become apparent until long after they are created and some may only appear for specific circumstances.

The industry response to this is to introduce unit testing and to make it automatic. This means that the project will automatically check that it is behaving correctly after each new addition to catch problems before they make it into the main branch. This is well worth implementing in your own projects, even if you are the only author and the scope is small. It is impossible to predict when, or where you will want to reuse parts of the project in the future or what expansions you will want to do. Debugging your code will use considerably more of your time, and is much more difficult, than writing it.

Cmake has a built in utility for running and managing the automated testing of your code, CTest. We will skip over how to create manual testing frameworks by hand as you should never do this and it's pretty ugly and jump straight to using one of the most common testing frameworks, GoogleTest (there are many others but google will do the job fine).

To add testing to our project we just add a directory `test` and add the following `CMakeLists.txt` file to it which downloads googletest and adds it to our project:

```

1 include(FetchContent)
2
3 FetchContent_Declare(
4     googletest
5     GIT_REPOSITORY https://github.com/google/googletest.git
6     GIT_TAG v1.13.0
7 )
8
9 # For Windows: Prevent overriding the parent project's compiler/linker settings
10 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
11 FetchContent_MakeAvailable(googletest)

```

Now we just need to add out tests which we do by writing some code that uses our library:

```

1 #include <gtest/gtest.h>
2
3 #include "../src/harmonic/harmonic.hpp"
4
5 TEST(harmonic, basic_operations){
6
7     EXPECT_EQ(harmonic::harm_add(20e0, 5e0), 4e0) << "Harmonic add failed basic
8         << operation";
9     EXPECT_EQ(harmonic::harm_add(5e0, 20e0), harmonic::harm_add(20e0, 4e0)) <<
10        "Harmonic add does not commute";
11 }

```

I strongly recoment that you call your testing files "test_library.cpp" so it is obvious what they are testing. Using the googletest framework we add tests with the command `TEST` followed by the name of the testing suite followed then the name of the test, here `harmonic` for the suite and `basic_operations` for the test itself. The tests are created using the two commands `EXPECT_*` and `ASSERT_*` with the difference being that the second abandons the rest of the test if it fails while expects continues. You would use `ASSERT_*` for something like checking the size of a vector before a loop which checks the entries because if the size is wrong the loop wouldn't make sense.

The types of test allowed can be seen here: <https://google.github.io/googletest/reference/assertions.html>. There are all the normal ones you would expect. Note for floating point arithmetic you have `EXPECT_FLOAT_EQ`, `EXPECT_DOUBLE_EQ` and `EXPECT_NEAR`. There is also `EXPECT_THAT` which can be used with a wide variety of matcher functions listed here: <https://google.github.io/googletest/reference/matchers.html>

Once we have created the test code we have to add it to the `CMakeLists.txt` in the testing folder

```

1 ...
2
3 add_executable(test_harm test_harmonic.cpp)
4 target_link_libraries(test_harm
5     PRIVATE harmonic_lib
6     GTest::gtest_main
7 )
8
9 include(GoogleTest)
10 gtest_discover_tests(test_harm)

```

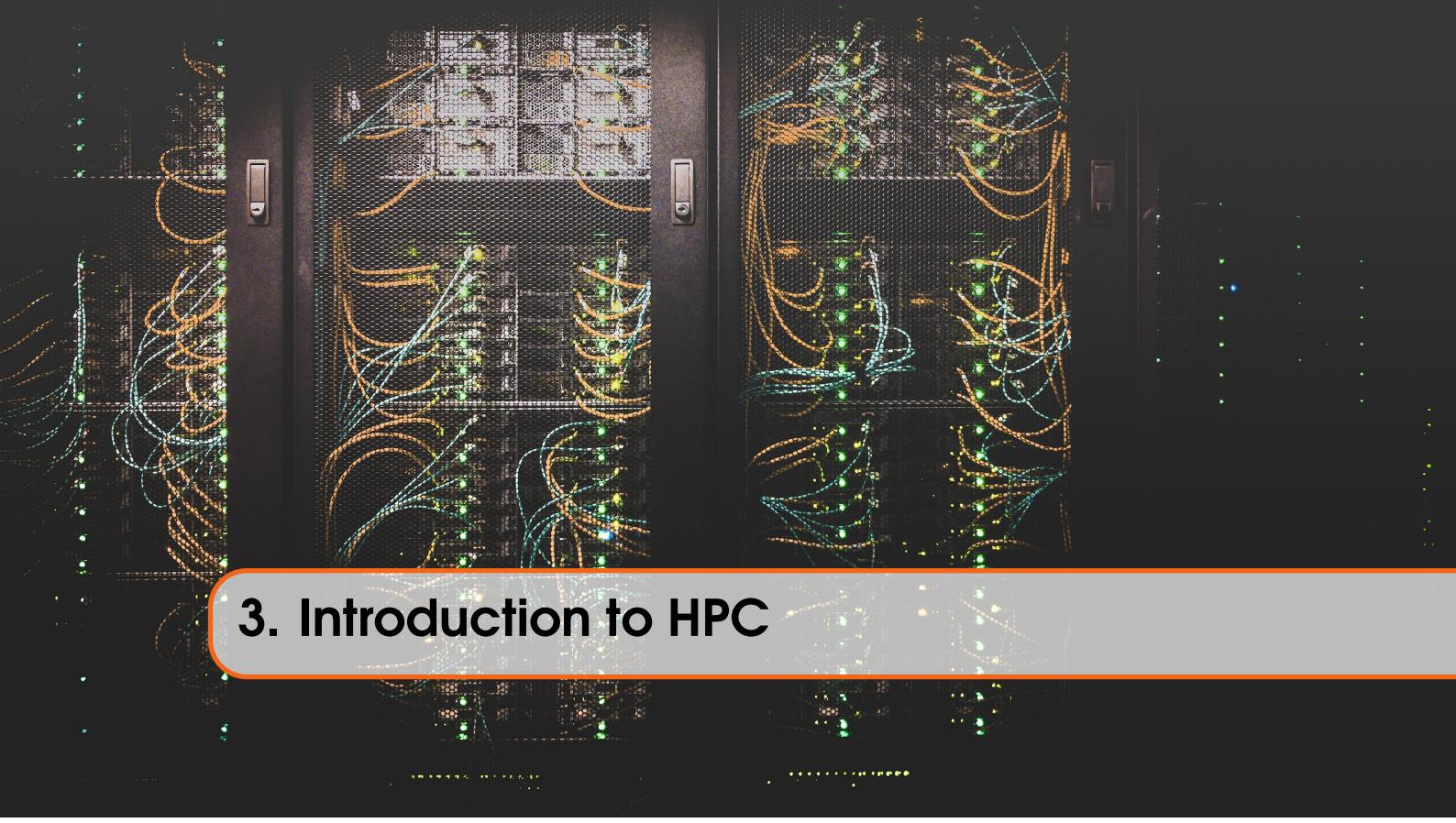
Now we just build the project normally:

```
1 $ cmake -S . -B build  
2 $ cmake --build build
```

And we can run the tests with:

```
1 $ ctest --test-dir build
```

Which will automatically discover all our tests and run them. Specific tests can be run with the /bash[-N] or -R options where the first takes an exact list and the second matches a regexp.



3. Introduction to HPC

Background

High Performance Computing for us means remote computing resources which are an order of magnitude more "powerful/performant/productive?" than your local machine. They can come in the form of: servers, clusters, supercomputers or accelerators.

Supercomputing is becoming an increasingly important topic in scientific research. Its growth is being driven by the exponential increase in data across all scientific research areas and also the availability of ever more powerful computing resources. In 2009 this was labeled the "Fourth Paradigm" of Data-Intensive Scientific Discovery (First three are: Direct Observation, Theoretical Modeling, Numerical Computation). With the explosion of Machine Learning techniques over the last decade we are creating the Fifth Paradigm of AI4Science where scientific discovery is driven by Machine Learning approaches. Both of the fourth and fifth paradigms are only possible on HPC resources.

The main use cases for scientific HPC are:

1. Computation of theoretical predictions
2. Simulation of physical systems
3. Data Analysis (sometimes referred to as HPDA)
4. Training ML models

Scientists who want to be effective in these areas increasingly need to master the task of writing efficient scalable parallel code. While typically scientists are HPC users who leave the complex task of creating and maintaining HPC systems to specialist HPC managers, in order to get best performance scientists need to gain an understanding of how HPC systems work. This has led to the new role of Research Software Engineer (RSE) who is a typically a scientist who has transitioned to becoming an expert on the implementation and optimisation of scientific codes on HPC systems.

The key descriptor in HPC is performance but what do we mean by this? The simple answer is the ability to do lots of Floating Point Operations Per Second (FLOPS). This is not actually a very useful benchmark for us. As scientists we are much more interested in productivity, not resource usage. We should aim for the maximum performance per unit of programming activity. The performance of our code will depend on many aspects of the architecture: memory size, latency,

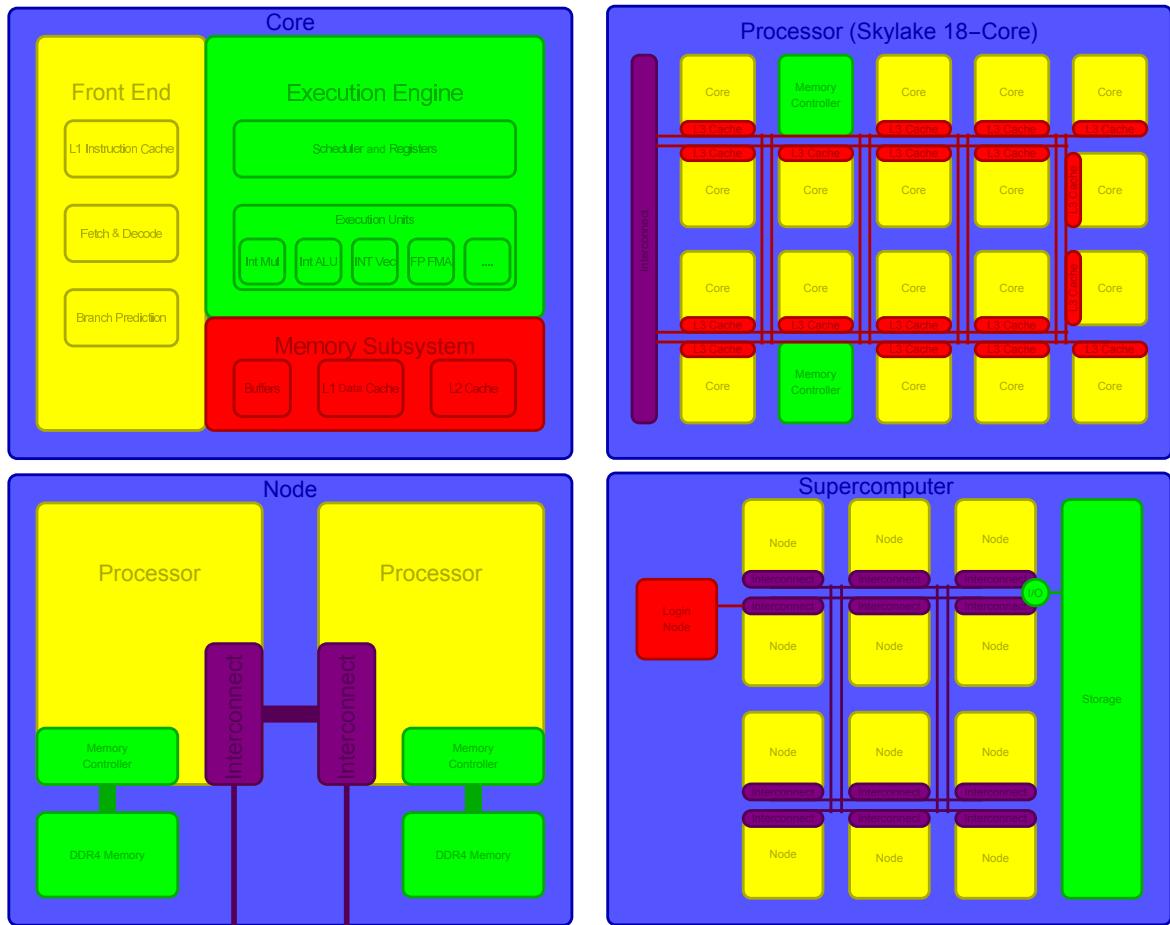


Table 3.1: Diagrams of the various layers of HPC architectures

bandwidth and locality; interconnect latency and bandwidth; accelerator access patterns; ease of programming paradigms. Supercomputer rankings are calculated for dense linear algebra problems solved using a specific library which may not be relevant to your application. We should also not chase performance at the expense of producing results. There is no point spending months optimising a code we will only use for a short period. Most gains are from finding optimal algorithms that work for the architectures available to you.

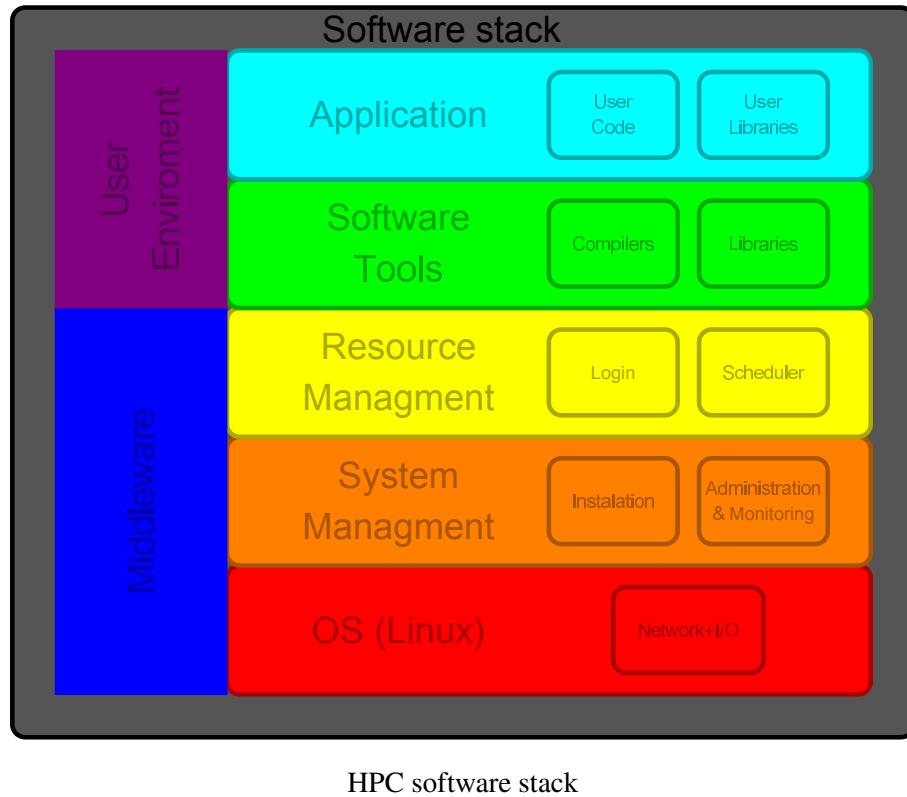
HPC dictionary

HPC resources are almost always remote to the machine you are working on. They are composed of the following pieces of hardware:

1. Nodes - which will contain processors (CPU/GPU/TPUs etc...), local memory and network cards. They are just like individual computers.
2. Interconnect - network which links the nodes together
3. Storage - Some kind of large disk space
4. Login Node - Special dedicated node that handles access to the machine.

Each node can have multiple XPU (X=C,G,T,...) devices so are described as "multiprocessor" (PU = Processing Unit). Each XPU will consist of multiple "cores" (smallest unit of computing). Generally processor=XPU=socket but sometimes processor=core (annoying!!). Please refer to the diagrams in 3.1 for a visual representation.

Supercomputers will also have the following software stack (please refer to ??):



1. OS - 99.999% of the time this will be Linux
 2. System Management - for HPC manager to control: users accounts, installation of libraries and software, and monitoring the machine
 3. Resource Management - Software to control access to, and use of, the hardware amongst users. Key in this is the scheduler which determines the jobs that can run on what nodes and when.
 4. Software tools - pre-installed software for users. Includes compilers and common scientific libraries.
 5. Application Layer - Consists of user installed libraries, scripts, and compiled executables.
- First 3 are the HPC system middleware and you cannot control them. The last 2 are controllable by the user and define the runtime environment for the application.

Access

HPC systems will almost always be accessed via ssh from the terminal (see setting up ssh config etc... in research computing lecture). Virtually all interactions with HPC will happen via the command line in the ssh terminal. Sometimes IDEs can manage some of this for you but you must know how to do all tasks from the command line if required. This is where you will control your environment and submit your jobs.

Scheduler

In order to manage a large number of users on a single system we need to have a scheduler which says who can use what resources, when and for how long. The scheduler will have queues set up which will have specific limits in terms of number of nodes requestable, maximum memory used and maximum walltime. If you violate these limits your job will either not run, or be terminated by the scheduler. You will submit jobs to queues using a terminal command to pass the scheduler a job script, which describes what resources you require, how many processes and threads to create, and for how long the job should run. Try to be accurate (+ some contingency). If you submit jobs from more resources than you need they will take longer to get through the queue and will

waste resources others could use. Schedulers will also assign users a priority (how this is done is endlessly configurable - see information pages for each specific system you use for more detail) which will typically decrease when you use the system and increase when you don't to fairly share the resources around the users.

Common schedulers are:

1. Slurm
2. Moab/Torque
3. PBS
4. Cobalt

We will look at Slurm as that is what is on Fawcett, all schedulers work in a similar way so knowing one is enough to allow you to adapt to any other. You can alternatively watch the following: Slurm Intro

A scheduler will have two parts that you will need to understand: a set of terminal commands to use to control and query your jobs and see the current status of queues; a format for job submission scripts to specify what resources you need and what executables to run.

Job scripts generally begin like this:

```

1 #!/bin/bash
2 #SBATCH --job-name=testjob
3 #SBATCH --mail-type=END,FAIL
   ↵ ALL)
4 #SBATCH --mail-user=email@something.ac.uk
5 #SBATCH --ntasks=8
6 #SBATCH --cpus-per-task=4
7 #SBATCH --nodes=2
8 #SBATCH --ntasks-per-node=4
9 #SBATCH --ntasks-per-socket=2
10 #SBATCH --mem-per-cpu=100mb
11 #SBATCH --time=00:05:00
12 #SBATCH --output=testjob.log
13
14 module load ...
15 srun {commands+executable}
16

```

Common Slurm commands:

1. `sbatch` - for submission \$ `sbatch myscript.sh`
2. `scancel` - obvious \$ `scancel <jobid>`
3. `squeue/sinfo/sstat/sacct` - view information on queues / nodes and partitions / running job statistics / recent job statistics
4. `scontrol` - detailed reporting on jobs or nodes eg: \$ `scontrol show jobid -dd <jobid>` but also controlling submitted jobs eg: \$ `scontrol hold <jobid>`, \$ `scontrol release <jobid>`, and \$ `scontrol requeue <jobid>`
5. `sprio` - view job priorities (or `squeue -P`)
6. `sview` - graphical display of system and jobs.
7. `salloc` - request allocation for interactive use (may not be disabled).
8. `srun` - request allocation for running an application (may not be disabled).

The most important are `sbatch` for submitting jobs to run on the system, and `squeue` to see its status.

Libraries

Mathematical libraries are controlled by the command `module` and are loaded by the user either in the job script or the terminal that submits the job (whose environment is used for the job at runtime).

Common libraries are:

1. BLAS/LAPACK/SCALAPAK - Linear Algebra
2. FFTW - Fast Fourier Transform
3. PETSC - ODE/PDE solver
4. Tensorflow - Machine Learning

Why should you use libraries? Simply because they will be much faster and more accurate than anything you will write. They are highly optimised and well tested, they will use sophisticated methods to reduce scaling and the libraries will have been built to optimise use of the available computational resources. They will often include thread level parallelism for free and, due to their ubiquity, they are highly portable.

The most common one you will use is BLAS which stands for Basic Linear Algebra Subprograms. It is written in FORTRAN77 (so short names and FORTRAN style array access). If you are using C/CPP then you can instead use CBLAS which has a C-style interface. BLAS contains simple vector-vector, vector-matrix and matrix-matrix operations (corresponding roughly to BLAS1, BLAS2, BLAS3). LAPACK adds more complex operations like linear solvers eigenvalue decomposition and singular value decomposition. ScaLAPACK extends this to distributed memory and parallel operation.

You will need to look at each package to see what it does and how to use it. These libraries are also wrapped into different packages, eg OpenBLAS (just BLAS/LAPACK) and intel's MKL library (which includes BLAS/LAPACK/ScaLAPACK, FFTW and PDE solvers plus DNN). The MKL is optimised to run on intel chips so performance is worse on other vendors hardware (there is an undocumented env variable to optimise MKL for AMD chips which is `export MKL_DEBUG_CPU_TYPE=5` but I've never tried this) OpenBLAS is hardware agnostic so is more portable but contains fewer libraries. AMD have their own version, AOCL (BLAS/LAPACK + FFTW plus RNG and crypto). There are many other libraries you can use but are less likely to be generally available eg NAG, GSL.

Library management is controlled by the `module` package. Common commands are:

1. `module avail`
2. `module list`
3. `module load`
4. `module unload`

As we stated before, you will have to load modules either to your environment via the command line after logging in, via your `.bashrc` file, or you can keep your login environment clean and add load the modules in the submission script itself. The last is best practice but the former makes sure the modules definitely load before submission.

4. Optimisation

Introduction

In this section we will look at compiler level optimisations that can speed up your code. This will cover two compiler level parallelisations that can make a very significant difference to your code performance - Vectorisation and Pipelining - which correspond roughly to parallelising data and parallelising instructions.

Warning

Optimisation is the last thing you should consider. First make sure that your code is readable and correct, properly documented and contains testing and validation automatically. Then you will need to profile the code to identify key areas that need optimisation where the code spends most of its time. You should also consider how much improvement is possible by using an adapted version of Amdal's law. This is simply the statement that if part of your code takes fraction p of the execution time then the maximum speed up from optimising it is a factor of $\frac{1}{1-p}$. If this factor is high enough to justify the effort you should then consider algorithmic and structural changes to the code as these will most likely give the largest improvements at minimal cost to readability and maintainability. Only after you have exhausted all of the above should you try the optimisation tricks in this chapter. Why? Because optimisation almost always reduces readability of your code making it harder for you to maintain and for others to use. It can also reduce the portability of your code by tying it to specific compilers or architectures depending on the options you choose.

Also: when doing any optimisation you should always create a new git branch specifically for this and keep your original code in the master branch. This way you can always go back to the original code if you need to and you can easily compare the performance of the two versions. You should also make sure that you have a good test suite for your code so that you can be sure that you have not broken anything when you make changes.

In this chapter we will only review the main options for optimisation with some simple illustrative examples in order to orient you to the field and give you some understanding of why certain practices are used. This is an area which can become very complex and murky as you

need to look very carefully at how your code is translated to machine code and how this affects performance which can be very non-trivial.

Luckily, if we write clean simple code often the compiler can perform many of these optimisations for us automatically which is the ideal case as we then avoid the ugliness which can come from implementing some of these optimisations by hand.

The key is to always write your code in a way that is task based and data driven. Specifically what we mean by this will become clear in the following sections.

Compilers

Compilers are designed to convert your code first to assembly language which is converted to machine code. Compilers are very sophisticated and during the conversion stage they perform significant optimisations automatically based on static analysis of your code. The goal of compilers is to implement your code in a way that will give the same answers but not necessarily with the process you have described. Frequently compilers will re-order or re-structure your instructions to make them more efficient for computers to execute. Mostly you should let the compiler worry about creating good machine code and you just worry about creating good "human code". That said there are some easy things we can do to help the compiler to do its job.

The first optimisation you should always do is simply to run with optimisation turned on. This is done with the optimisation flag `-O` which has several options:

1. `-O0` :: No optimisation, just transliterate your code exactly.
2. `-O1` :: Reduce size and runtime without doing anything that would take sophisticated analysis.
3. `-O2` :: Perform all optimisations except those that do not involve a space/speed trade-off.
4. `-O3` :: Do everything the standard allows.
5. `-Ofast` :: `O3` but including stuff that breaks the standard. (`-O3` is preferred over this in virtually all cases)
6. `-Os` :: Use `-O2` but without anything that will create a larger executable, can sometimes be surprising

Next we should tell the compiler to compile the program for the specific processors it is going to run on. This is done with `-xHost` for `icc`, or `-march=native` for `gcc`. This obviously reduces portability of the executable which in HPC we almost never care about as we will rebuild the code on each machine we use.

We can also optimise our code based on its runtime performance. The standard compilers; `icc`, `gcc`, and `clang` all support the generation of runtime reports which can be used to optimise subsequent compilations:

```

1 $ gcc -fprofile-generate
2 % then run the code and rebuild with
3 $ gcc -fprofile-use
4
5 $ icc -prof-gen
6 % then run the code and rebuild with
7 $ icc -prof-use

```

All of the above can have significant improvements without any code changes so should be used in every project, every time.

In order to see what the compiler is doing with your code you can use the excellent website: GodBolt which will show the generated assembly code from the input C++ for any compiler (you cannot see the effect of `-xHost` or `[-march=native]` as you cannot select hardware). To generate these reports at compilation, you can include the options:

```
1 $ gcc -S -fverbose-asm
2 $ icc -S -fsyntax-only
```

which generate annotated output files of the assembly code generated. We will be looking at some of the assembly code for simple examples to understand the optimisations in the following sections.

Easy wins

Once you have enabled the compiler options there are some easy wins you can get from using good programming practices. We will describe some of the more common mistakes people make in this section so you can avoid them.

Storage

The first is storing variables in the correct types. We have seen how memory access is a major bottleneck for code performance so ensuring that we give the compiler as much information on where to store our variables is very important. We have the following qualifiers we can add to memory assignment:

1. `extern` :: Global variable that exist for the entire life of the program
2. `auto` :: local variable created on the stack for a limited scope then destroyed. Must be initialised
3. `register` :: Suggest that the compiler place these directly in the register
4. `const` :: Variable that never changes
5. `volatile` :: Variable that can be changed from outside the program
6. `restrict` :: Memory that is only accessible via the pointer.

Use `extern` for global variables. `auto` allows the determination of the variables type from the initial assignment so can save on type conversions, for example `int a=1.0;` or `double x=1;` both require a type conversion to assign the value. `register` is a suggestion (the compiler is free to ignore you) that the variable will be re-used soon after so keep them handy. `const` is for constants, if the compiler knows that they will never change it can just do a straight replace everywhere removing the need for references. `volatile` tells the compiler to watch out. `restrict` is useful because in CXX it is possible to have two pointers that access the same memory, called "aliasing" (and forbidden in FORTRAN). This means that the compiler has to be careful with optimisation incase two variables point to the same location in memory, which would break some optimisations. For example:

```
1 *a += *b;
2 *a += *b;
3 % is not the same as
4 *a += 2 * *b;
5 % if a points to the same location a b
```

`restrict` promises to the compiler that this is not the case so it can be more aggressive in its optimisations

Functions

Functions are much more expensive than intrinsic operations. These include: {sqrt,pow,division} You should try to avoid them as much as possible. For example:

```

1 y = pow(x,2);
2 % vs
3 y = x*x;
4 % or
5 y = x/10;
6 % vs
7 y = x*0.1;

```

In the second example we actually have a double penalty of a type conversion and a floating point division, which is 3 times slower than multiplication but also affects throughput for an extra penalty. Also note that division does not vectorise (see later) so there is a parallelisation penalty. On the other hand we can store 10 exactly but notn 0.1 so the answers may change slightly between the two lines. Many compilers will automatically optimise y/x to $y * 1/x$ by calculateing the reciprocal using a fast approcimation. This can be explicitly requested with the compiler option `-ffast-math` for gcc, icc does it by default.

Loops

Loops are almost always where you have to spend the most time optimising due to the simple fact that the code in them is repeated many times. This means that we should try to avoid as much computation in the innerloop as possible. For example:

```

1 for(i = 0; i < Ng; i++){
2     for(j = 0; j < Ng; j++){
3         for(k = 0; k < Ng; k++){
4             dist2 = pow(x[p] - (double)i/Ng - half_size, 2) + pow(y[p] -
5                         ↪ (double)j/Ng - half_size, 2) + pow(z[p] - (double)k/Ng - half_size,
6                         ↪ 2));
7         }
8         if(dist2 < R2){
9             % do something;
10        }
11    }
12 }

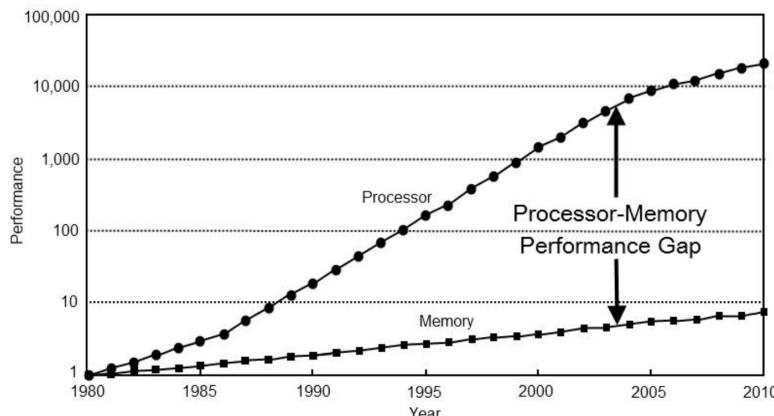
```

Could be changed to:

```

1 double register Ng_inv = 1.0 / Ng;
2 for(int i = 0; i < Ng; i++) {
3     double dx2 = x[p] - (double)i * Ng_inv - half_size;
4     dx2 *= dx2;
5     for(j = 0; j < Ng; j++) {
6         double dy2 = y[p] - (double)j * Ng_inv - half_size;
7         dy2 *= dy2;
8         double register dist2_xy = dx2 + dy2;
9         for(k = 0; k < Ng; k++) {
10            double register dz = z[p] - (double)k * Ng_inv - half_size;
11            double register dist2 = dist2_xy + dz*dz;
12            if(dist2 < Rmax2){
13                % do something with sqrt(dist2);
14            }
15        }
16    }
17 }

```



Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau

Memory Wall

Here we have done several things:

1. Precomputed division of $1/Ng$
2. Replaced pow functions
3. Moved calculation that does not involve k out of the inner loop (called "Hoisting" and sometimes done by the compiler automatically)
4. Registered frequently used variables
5. Left the creation of local variables ($dx2$ etc...) to inside the loops to limit their scope

We should also remove functions from loop conditions, eg: don't use `while (i < strlen(string))` if the string length is not going to change

Note compilers optimise int operations much better than floating point as on computers. FP operations are *not* associative on digital hardware so the compiler will not re-order operations (as the compiler is wary that you have ordered operations in a specific way for accuracy) which can impede optimisation. If this is not the case you can tell the compiler using the flags:

```
1 $ gcc -funsafe-math-optimizations
```

Intel's compiler is naturally more aggressive and this is turned on by default at -O2 and above. To turn this off use

```
1 icc -fp-model source
```

Cache

Modern high performance computing is strongly affected by the "memory wall" because processor performance improves at a faster rate than memory bandwidth. Without sufficient care many codes will stall as the processor becomes starved of data to process. Currently the solution is to create a cache hierarchy which gets smaller and faster the closer you get to the core. This allows us to put the data we need to work on close to the core so it can be accessed fast while keeping the other data in the larger slower regions of memory. The processor then will move the data up and down the hierarchy as needed.

Here are some details for the speed of memory access for the Intel Sandy Bridge processor

Level	Size	Latency (cycles)
Register	256 B	1
L1	32 KB	4
L2	256 KB	12
L3	3 MB	28
RAM	16 GB	28 + 56ns (185)

When the processor requests data it first looks in the L1 cache. If the data is not there then we have a "cache miss". Suppose that we had only L1 and RAM then accessing a variable in L1 costs 4 cycles and accessing a variable in RAM costs 213 cycles. So for different cache miss rates we have the following data rates:

1. 100% -> average of 4 cycles
2. 99% -> average of 6 cycles
3. 97% -> average of 10 cycles

So very few misses can be very costly. With L1,L2,L3 and RAM this is much better

1. 100% -> average of 4 cycles
2. 99% L1 + 100% L2 -> average of 4.08 cycles
3. 97% L1 + 100% L2 -> average of 4.24 cycles
4. 90% L1 + 97% L2 + 99% L3 + 100% RAM -> average of 4.85 cycles

so we can see that the hierarchy is very good at removing the penalty for memory access provided that we can keep the caches full of the correct data. But from the first example we can see that if we only have 1% of our data having to be fetched from RAM then our memory performance degrades by a factor of 2.5!

You cannot control (and wouldn't want to!) the memory management so how can we ensure that the data we want is in cache? The processor tries to guess what data will be needed based on locality. This means both temporal and spatial - so if something is used then; it will likely be used again soon, and its neighbours will likely be used soon.

You can see how your code is performing on cache misses by profiling with cachegrind which is part of the valgrind package. First you have to profile the programme (with debugging turned on, -g but also with optimisation), then process the output to annotate the code with the results:

```
1 $ valgrind --tool=cachegrind executable
2 $ cg_annotate <filename>
```

This will produce simulated output of cache misses from L1 which is usually enough to let you know where you have issues.

What can we do to optimise cache performance? Firstly we should try to do all calculation on each piece of data consecutively. This means we should avoid writing code like this:

```
1 for(i=0;i<npix;i++){
2     map[i] = map[i]*mask[i]
3 }
4
5 for(i=0;i<npix;i++){
6     map[i] = map[i]-level*foreground[i]
7 }
```

Compilers will try to merge these two loops as this example is quite simple, but without optimisation we would have to load the data in `map` twice which is much more expensive than putting it in one loop.

We should also make sure that all our data is stored and accessed in a contiguous manner. Compare:

```

1 std::vector<int> A,B;
2 A.reserve(N_col*N_row);
3 B.reserve(N_col*N_row);
4
5 ...
6
7 for(row=0;row<N_row;row++){
8     for(col=0;col<N_col;col++){
9         A[N_row*row + col] = 5.2*B[N_row*row + col]+3.0;
10    }
11 }
12 // Vs
13 for(col=0;col<N_col;col++){
14     for(row=0;row<N_row;row++){
15         A[N_row*row + col] = 5.2*B[N_row*row + col]+3.0;
16    }
17 }
```

The first will be much faster as we are accessing the data in order in memory. In the second we are accessing the memory in a strided pattern so we are violating locality (this is more expensive in terms of instructions, and will fail to vectorise, so is terrible all round).

Similarly we see the same issues when we use vectors of structs/classes which inevitably result in strided access.

When using large arrays we can often improve performance with **cache blocking**. This is when we break up one loop into chunks so we can keep it in the L1 cache. Cache actually has a structure where memory is organised into sets of n lines where each line contains m bytes. Consider the simple example of looping over a 8 element vector twice when L1 can only contain 4 elements in two lines of 2:

```

1 std::vector<int> A;
2 int size = 8;
3 A.reserve(size);
4 for(i=0; i<2; i++){
5     for(int j=0; j<size; j++){
6         // something with A[j]
7     }
8 }
```

The code will load the line of data into L1 whenever a variable is needed. This will have the cache miss pattern: MH MH MH MH MH MH MH as we load two values as a line so first is a miss, then the next will be a hit, then we need to load the next two values in the next line and miss again. Consider the following rearrangement:

```

1 std::vector<int> A;
2 int size = 8;
3
4 A.reserve(size);
5 int block_size = 4;
6 int block_num = size//block_size;
7
8 for(k=0; k<block_num; k++){
9     for(i=0; i<2; i++){
10         for(j=k*block_size; j<(k+1)*block_size; j++){
11             // something with A[j]
12         }
13     }
14 }
```

Now with some thought we can see that as block size is the same as the L1 cache the cache miss pattern is now: MH MH HH HH MH MH HH HH as we load the two lines into the cache then use them again so only have hits. This means that we have reduced the cache miss % from 50% to 25%.

Sometimes we have to access arrays in a strided pattern (ie taking each nth element) for example if we are transposing a matrix we will have to either read or write in a strided pattern. There is one odd issue that you can run into with is related to the cache memory structure and can cause significant performance impact (up to a factor of 10). This occurs when the stride is a large power of 2. As the processor needs to index memory in the cache and RAM very quickly (remember that L1 has a latency of 4 cycles so we need to be able to decode memory pointed in less than this) it uses maps data between the two by breaking the cache into small blocks (sets of lines) and mapping these blocks onto specific regions of memory. This means that in RAM regions which are separated by a whole number of blocks map to the same region of cache. So, when you stride by a large power of two each block you access wants to go to the same cache location so you only use one block of cache. This massively reduces the effective cache size so badly affects performance. To see this compare timing for the following two code samples:

```

1 for(i=0; i<N; i+=256){
2     A[i] = i;
3 }
4 //Vs
5 for(i=0; i<N; i+=257){
6     A[i] = i;
7 }
```

This issue is known as “Cache Resonance” and happens more often than you might think given how frequently scientific data sets come in multiples of large powers of 2.

Finally if you are using structs/classes in your code it is important to order the variables properly. For example for the linked list struct:

```

1 struct my_node
2 {
3     double key;
4     char my_data[300];
5     my_node *next_node;
6 }
```

To get to the field `next_node` we have to pass over 300 characters. It would be more efficient to reorder the last two fields so they are in the order in which they are most commonly accessed.

Even better, you should consider separating the meta-data from the data like this:

```

1 struct my_node{
2     double key;
3     my_node *next_node;
4     void *my_data;
5 }
6
7 struct my_data{
8     char data[300];
9 }
```

Structs/Classes always mess up data access and cache coherence so carefully consider their use.

Branches

A Branch occurs in your code whenever there is more than one possible set of instructions that could be executed. They occur whenever you have either `if` or `for` statements (or similar) in your code. When converted to assembly these create "jump" instructions as the compiler is not sure which instructions should be run after it so the code will jump to one or the other depending on the runtime conditionals evaluated (This is essentially the FORTRAN `GOTO` reappearing as that is what the processor actually understands).

The processor will try to guess the correct instructions and load them into the instruction L1 cache, just like it tries to guess the next data you might want and loading it to the L1 data cache. If the compiler gets this wrong then the core will have to wait for the correct instructions to be fetched. Typically compilers will guess the TRUE is the most likely and load that so we need to think about "code locality". The cost of mis-predictions can be very large. Just like cache misses, branch misses can be calculated with profiling using cachegrind.

To avoid this try to be sparing regarding conditions and move them as far out of loops as possible. For small if statements it can be faster to just run the code for both options then compare and write the correct output at the end as this will not involve any branching. The compiler will try to do this as much as possible, ie something like (but in machine code where the last can be implemented in a conditional move):

```

1 if (a<b) {
2     c=a+b;
3 }else{
4     c=a-b;
5 }
6 // becomes
7 tmp1 = a+b;
8 tmp2 = a-b;
9 c = (a<b) ? tmp1 : tmp2;
```

This is known as masking and is a very useful technique.

Sometimes for small loops you can be clever by using the fact that true is 1 and false is 0:

```

1   for(i=0;i<n;i++){
2     if(a<b){
3       sum += a;
4     }else{
5       sum += b;
6     }
7   }
8 // becomes
9   for(i=0;i<n;i++){
10    log = a<b;
11    sum += log*a + (1-log)*b
12  }

```

which has no jumps. In this case the compiler may have performed equivalent optimisations for you so you may see no benefit. For more complex examples it could be worth a try. Sometimes the if statement outcome is predictable so you can just separate the code, eg:

```

1 for ( j = 0; j < N; j++ ){
2   for ( i = 0; i < M; i++ ){
3     if ( i > j )
4       matrix[j][i] = 1.0;
5     else if ( i < j )
6       matrix[j][i] = -1.0;
7     else
8       matrix[j][i] = 0.0;
9   }
10 // becomes
11 for ( int j = 0; j < N; j++ ){
12   for ( i = 0; i < j; i++ ){
13     matrix[j][i] = -1.0;
14   }
15   matrix[j][j] = 0.0;
16   for ( i = j+1; i < M; i++ ){
17     matrix[j][i] = 1.0;
18   }
19 }

```

So no conditionals required.

Finally if you have lots of ifs you should try to use other structures like switch case.

The reality is that most of the time you can restructure your code to greatly reduce the use of conditionals which can have a very significant effect on performance.

Pipelines

There are two types of parallelism possible at the processor level, data and instruction. Pipelining concerns the second, Instruction Level Parallelism (ILP). This is when we deliver multiple instructions to the core in each clock cycle as they are executed using different parts of the chip so can run concurrently. This allows us to overlap the instructions increasing throughput.

There are 4 basic types of instruction

1. Fetch
2. Decode
3. Execute
4. Write

Each of these can be broken down into several smaller sub-tasks with the goal of getting down to a set of 1 clock cycle instructions blocks (this sub dividing is called super-pipelining). Once we have these we can aim to overlap them on all available hardware nodes (fetch, decode, int ops, fp ops, test & branch, memory ops) to maximise performance.

There are three main hazards which can block ILP

1. Structural
2. Data
3. Control

Structural hazards occur when there are not enough hardware units to execute the instructions given to it, or different operations have very different timings (division can be much slower than multiplication). Data hazards occur when there are data dependancies which limit concurrency, ie step x needs the output of step y. Control hazards occur when there is branching so we can't predict the next instructions accurately.

Pipelining is handled by the compiler so you cannot directly control it (enabled from -O2 and above)

You can help the compiler do this with two optimisations, neither of which should be used lightly as they affect readability. The first is loop unrolling where you write code that repeats the code in loops to reduce the iterations required, eg:

```

1 for (i=0;i<N;i++){
2     S += A[i]+B[i];
3 }
4 // becomes
5 for (i=0;i<N;i+=4){
6     S1 += A[i]+B[i];
7     S2 += A[i+1]+B[i+1];
8     S3 += A[i+2]+B[i+2];
9     S4 += A[i+3]+B[i+3];
10 }
11 S = S1+S2+S3+S4;

```

Here the cumulative sum makes each loop look dependent so the compiler can't optimise it. By breaking it into 4 parts the compiler can see that they are separate and can be calculated in parallel.

Compilers will do this automatically but sometimes it will struggle to see the optimisation if there appear to be dependencies like above, or if there is branching, or if we are dealing with floating point operations as it is not safe to reorder them. You can force this by using the -funroll-loops flag which will force the compiler to unroll loops. This can be effective but can also be very very bad as it makes the instructions longer and can trash your cache. It's not included in any of the -O optimisations for a reason. You can also gain some performance by separating the sum and the assignment using local variables as this makes clear that the instructions are separate (only small gain), ie:

```

1 for (i=0;i<N;i+=4){
2     double tmp1 = A[i]+B[i];
3     double tmp2 = A[i+1]+B[i+1];
4     double tmp3 = A[i+2]+B[i+2];
5     double tmp4 = A[i+3]+B[i+3];
6     S1 += tmp1;
7     S2 += tmp2;
8     S3 += tmp3;
9     S4 += tmp4;
10 }
11 S = S1+S2+S3+S4;

```

Loops, Prefetching and Vectorisation

Finally we will look at the other processor level parallelisation, Vectorisation. Most modern cores have special logic to allow them to perform the same operation on multiple data elements at the same time. This is called SIMD (Single Instruction Multiple Data). The compiler will look for loops which can be vectorised and will do so automatically in most cases but sometimes needs some help. Main blockers are the usual suspects, branching, complicated referencing or possible dependancies.

This is most useful for vector operations (obviously) using either vectors or arrays. The requirements for vectorisation are generally:

1. Iterations are known at runtime. (no `break` etc...)
2. No complex branching (`switch` or `if` statements which can't be masked)
3. Only works on the innermost loop so order operations accordingly
4. No function or I/O calls other than vector functions or ones that can be inlined
5. No data dependancies (ie no cumulative sums)
6. Contiguous memory access (possible if not, but much slower)

You can see if the code has vectorised by generating vectorisation reports using the `-fopt-info-vec` flag for `gcc` or `-qopt-report=2 -qopt-report-phase=vec` for `icc`.

If something you think should have vectorised hasn't then there are several additional things you can do. Unrolling can be helpful in exposing parallelism where there are apparent dependancies, like with the cumulative sum in the previous section and using the `restrict` will ensure the compiler that that the arrays do not overlap. You can also tell the compiler to ignore vector dependancies for specific loops using the pragma:

```

1 #pragma ivdep
2 for (i=0; i<N-M; i++) {
3     x[i] = x[i+M] + s1;
4 }

```

Intel's `icc` give you the more insistant `##pragma vector always` which will vectorise if it is technically possible and `##pragma simd` which will vectorise even if it is not safe to do so. Both obviously reduce portability.

If this fails then it is time for more heavy duty options.

You can use compiler specific extensions like the `gcc` vector extension (which is supported by `icc`) `****` for `gcc` or `_mm*` for `icc` which reduce the portability of your code. These allow FORTRAN style array operations like:

```
1 typedef double v4df __attribute__ ((vector_size (32)));  
2  
3 typedef union {  
4     v4df v;  
5     double d[4];  
6 } v4df_u;  
7  
8 v4df_u a,b,c;  
9  
10 for (i=0;i<4;i++){  
11     a.d = i;  
12     b.d = 4.0-i;  
13 }  
14  
15 c.v = a.v+b.v;
```

To increase performance we can ensure that the arrays that we are using are aligned with memory lines as they are faster to read. This is done using the following:

```
1 __attribute__((__aligned__(32)))
```

which will ensure that the array is aligned at memory boundaries. If you do this it is also useful to pad the arrays so that they also end on a memory boundary.